15-411 Lab 5 Report: Interligua

Scott Mionis
Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania
smionis@andrew.cmu.edu

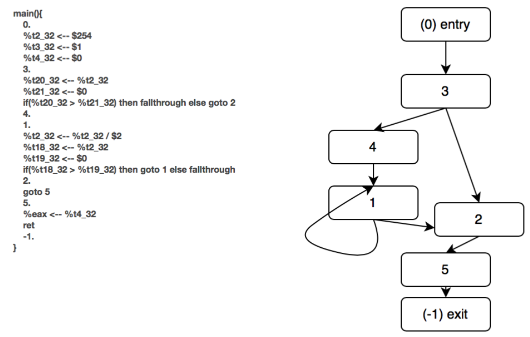
Akshath Jain
Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania
arjain@andrew.cmu.edu

***Abstract*—In this report we summarize the various optimizations we integrated into our OCAML Compiler framework for Lab5. These optimizations are best divided into one of several categories. Analysis passes we implemented consist of Control Flow Graph construction, SSA, Dataflow analysis and Dominator Trees. Optimization Passes we implemented consist of various peephole optimizations performed to speed up our original Lab3 and 4 compilers, SCCP, PRE, DCE, Function Inlining and TCO. also of note are the interactions between these passes, predictable and not. We then analyze the types of programs our optimizations perform well on, and perform poorly on. We end by summarizing our results and describing future work to be done.**

***Index Terms*—Compilers, 15-411, C0, Optimizing Compilers**

All statistics evaluated on the Notolab Servers (contact your local CMU representative for technical specifications) unless otherwise specified. Average statistics presented evaluated over the 15-411 benchmark tests unless otherwise specified

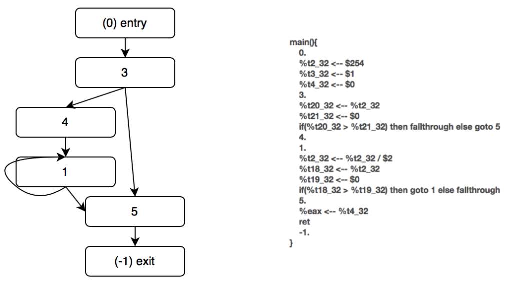
Analysis Passes: CFG. One major change to our original Lab4 framework was adding a complete and standalone CFG module. This module takes the output of our codegen module (which is a linear program interspersed with labels) and converts it into a control flow graph that we can do post-processing on:



```
main(){
    0.
    %t2_32 <-- $254
    %t3_32 <-- $1
    %t4_32 <-- $0
    3.
    %t20_32 <-- %t2_32
    %t21_32 <-- $0
    if(%t20_32 > %t21_32) then fallthrough else goto 2
    4.
    1.
    %t2_32 <-- %t2_32 / $2
    %t18_32 <-- %t2_32
    %t19_32 <-- $0
    if(%t18_32 > %t19_32) then goto 1 else fallthrough
    2.
    goto 5
    5.
    %eax <-- %t4_32
    ret
    -1.
}
```

Fig. 1. CFG Formation

Once a CFG is constructed, we perform a variety of simplification passes. Firstly, we perform 'CFG Simplification' which entails detecting basic blocks solely consisting of branches and returns, and deleting them by pulling the relevant branches into the parent nodes. While it is harmless to pull in a GOTO, this actually turned out to be a trade-off between code size and speed for the return case. This is because if you have multiple blocks that all branch to a block that immediately returns, you can save on speed by simply having each of the predecessors return, but you lose on code size because now each predecessor must duplicate the pop/stack frame reset code.



```
main(){
    0.
    %t2_32 <-- $254
    %t3_32 <-- $1
    %t4_32 <-- $0
    3.
    %t20_32 <-- %t2_32
    %t21_32 <-- $0
    if(%t20_32 > %t21_32) then fallthrough else goto 5
    4.
    1.
    %t2_32 <-- %t2_32 / $2
    %t18_32 <-- %t2_32
    %t19_32 <-- $0
    if(%t18_32 > %t19_32) then goto 1 else fallthrough
    5.
    %eax <-- %t4_32
    ret
    -1.
}
```

Fig. 2. CFG Simplification - Used to eliminate extra blocks created with critical edge splitting for SSA and PRE

We also support Critical-Edge Splitting which is necessary for both PRE and SSA. Critical-Edge Splitting introduces a lot of new and empty basic blocks temporarily, but allowed for our PRE and SSA algorithms to run more efficiently (and simply run at all). CFG Simplification is then done after both SSA and PRE to delete any critical edge blocks that have not been filled with meaningful computation (such as loop preheaders after PRE). Runtime analysis of these operations is provided at the end of the document.

Performance Evaluation
Git Commit : 1e66d0ccb4e41feebe9ddd0fbec880f50a1b1af6
Lab5Benchmark Submission ID: 6
Relative Increase in Avg Benchmark Speed: .03
Relative Increase in Avg Code Compression: .01

Analysis Passes : SSA and Dominator Trees Construction of SSA happens in two passes. The first pass uses the Control Flow Graph to construct a Dominator Graph, which is simply a CFG but augmented with Block-level information about dominance, immediate dominance, and the dominance frontier and discarding program information. We used the Dominance frontier algorithm from lecture which required minimal modification [1]. The dominance frontier is the most important metadata field of this step, as it is crucial to SSA. To convert our CFG into an SSA-form CFG, we take an input CFG and Dom Graph, and systematically place phi functions and rename temporaries according to the Semi-Pruned SSA algorithm from lecture [1]. The output is an SSA Graph, which is the original CFG with program content modified with temporary renaming and phi function insertion. To convert out of SSA, we started by renaming all the temporaries in the code that had different subscripts, and inserting moves to resolve Phi functions. We convert out of SSA before register allocation, so we did not have to deal with Spartan Transfer Graphs to take it out of SSA. However, since most of our SSA optimizations did not take out code out of CSSA, we ended up being able to simply drop subscripts on temporaries, remove phi functions, and still maintain correctness. SSA itself was only worth it due to DCE and SCCP, and took a significant fraction of the compiler runtime as seen at the end of the document. Also of note is it's potential interaction with PRE, which is explained in the PRE section.

Performance Evaluation
Git Commit : ae8c24758bb7f003a2668f910169c1a19b459708
Lab5Benchmark Submission ID: NA : see DCE and SCCP
Relative Increase in Avg Benchmark Speed: .00
Relative Increase in Avg Code Compression: .00

Analysis Passes : Dataflow Framework Another major change to our Lab4 compiler was modifying the dataflow framework to be more generalized, and to operate on expressions as well as temporaries. Dataflow is still used in liveness analysis (Backwards May, facts = temporaries) [2]. However we also needed to adapt the framework to support PRE dataflow passes which act on expressions and support a variety of directions and meet operators. We ended up making a separate copy of dataflow in pre.ml, since it was easier than generalizing fact objects for us, but in retrospect we could have done much more with alias analysis, etc... had we had a more general framework.

Performance Evaluation
Git Commit : 796a41b3a1e00f37814004f2a5356c5284b16243
Lab5Benchmark Submission ID: NA : see PRE
Relative Increase in Avg Benchmark Speed: .00
Relative Increase in Avg Code Compression: .00

Optimization Passes : Cleaning up Lab3 and 4

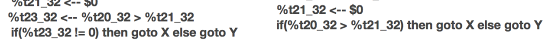
a) Removing extra moves and Temporaries We unfortunately started with a codegen module that was relatively overeager when it came to placing arguments in temporaries, and because of this we had final code consisting of long strings of moves that achieved practically nothing. To remedy this, we modified the logic in codegen to only move arguments into temporaries if they were not already in temporaries. This problem is also solved with copy and constant propagation, but we decided handling it earlier meant smaller code sizes for the rest of the compiler and significantly sped up the downstream passes. This optimization had significant implications for other optimizations. We found that PRE had almost 0 benefit before this change was made, because equivalent expression were being obfuscated by the fact that the temporaries were renamed. Copy propogation would also solve this problem, but this module was never implemented efficiently. This also significantly reduced the size of the code, making the runtime of PRE, SSA, and Liveness Analysis a bit more tenable. (Quantitatively, we measured about a 3 second overall change on the larger l4-new benchmarks (the team Vulcan suite). evaluated on a 2018 Macbook Pro).

b) Basic Block reordering When generating code for if statements, we reorder basic blocks in our codegen module to make maximum use of the fallthrough case. To achieve this, we also implemented an additional parameter that is passed around with the CFG. This parameter is an integer list, and simply denotes the order of the basic blocks in the file, so that we maintain the ordering for fallthrough branching cases. This ”porder” or print-order parameter is used by the format_reg_output function in liveness.ml to finally flatten the CFG basic blocks, in the right order, before passing the result into the register allocator. We would not have needed the print order had we eliminated fallthrough cases, but this optimization primarily had the effect of improving both runtime and code size, evaluated on the benchmark tests, as declared below.

c) Removing uses of spill register R10 In our original compiler, we were drastically overeager to spill to the R10 spill register to avoid checking whether both operands to a x86 code double were StackSlots, or alternatively conflicted. We now only use it when absolutely necessary, such as when we are adding two StackSlots (since most instructions do not support both source and destination as memory locations). This optimization did not affect any downstream passes since it happened last, but it did cut down code size quite a bit, as seen below.

d) Evaluating constant expressions Given an expression purely made of constants, we evaluate this using OCaml's builtin Int32 module, and convert it into a move before passing the code into the backend. This reduced both code-size (reducing a large L1 statement into a single move was valuable), speed of the executable (for costly operations like modulus, shift, or divide), and sped up PRE significantly (by approximately 0.2 seconds estimated over the benchmarks, run on a 2018 Macbook Pro) since PRE no longer had to check and compare purely constant expressions like '3 + 0' or '5 ¡= 32'.

e) Reducing Compares While more generally, booleans are expressed in our code as 1 and 0 bits, we optimized branching on comparisons by adding an IfCompare instruction:



```
%t20_32 <-- %t2_32
%t21_32 <-- $0
%t23_32 <-- %t20_32 > %t21_32
if(%t23_32 != 0) then goto X else goto Y
```

```
%t20_32 <-- %t2_32
%t21_32 <-- $0
if(%t20_32 > %t21_32) then goto X else goto Y
```

Fig. 3. IfCompare Instruction Simplification

Normal 'If' statements branch by evaluating whether the condition is boolean 1 or 0 (and explicitly checks). But IfCompare branches based off of the original compare result, which proved extremely useful on the benchmarks, as seen below.

f) Fixing Function Arguments In Lab3 and 4, we constructed a 'packing' and 'unpacking' scheme for putting arguments into the properr edi, esi, ... registers and taking them out at the beginning of a function. However, if one of the target destinations were any of the argument rergisters, the unpacking step would often clobber a value before it was loaded. To circumvent this we explicitly disallowed any argument or parameter registers to be any of the set [edi, esi, ...].

This was bad because a) it increased register pressure and b) because of this increased register pressure, more callee-saved registers were placed on the stack. Eliminating this requirement by inserting proper moves at the beginning and callsites before register allocation was used, but we did not see benefits until coalescing was implemented, since now edi, esi, .. could be coalesced with their target destinations.

g) Fixing Safety Checks We applied the IfCompare instruction to simplify array bounds-checking. Additionally, instead of using the divide instruction to cause a div-by-zero in the case of an improper shift, we introduces the CmpError instruction which performed a comparison and asserted the proper error if violated. Since error checks were inserted liberally, this had a large affect, as seen below.

h) Memory Addressing Instead of performing add and mult instructions to form a basepointer, and then using the single (ptr) addressing mode, we introduced the Lea instruction whihc both cut code size and was faster than 2 separate arithmetic operations.

Performance Evaluation
Git Commit:
a) 36123194be3b73bff50dc1c044684240d0a14a04
b) dd5d8cde6ff41f408cadc7e624afd324e7d57148
c) 694f07f22c81c5a46ee0cb101e2c10e9278276d0
d) 4a4b50a7c5423c1db674899160b25e54707da51b
e) 1a9a968af80ea41c970264aaed493635c88f8664
f) 5ba2c306fcdc1be9f576d219701e7bd957e682b5
g) d6a0ad1754e8674e780c42c9fbfe8f8a4efd7814
h) 8bfb0e9ef406ae365fbcca14808ab6ca4d0e29ff
Lab5Benchmark Submission ID:
(many were implemented in batches, and so have the same submission ID. Performance was evaluated locally for such occurrances on a 2018 Macbook Pro running High Sierra)
a) 4
b) 5
c) 4
d) 17
e) 56
f) 58 (but also didn't have SSA enabled, and so that is why score went down)
g) 34
h) 37
Relative Increase in Avg Benchmark Speed:
a) .1 (our original Lab4 was very ugly in this respect, this was more of a shift from timeout to not timeout)
b) .07
c) .2 (our original Lab4 was very ugly in this respect, this was more of a shift from timeout to not timeout)
d) .05
e) .12
f) .015
g) .04 (evaluated locally, we encountered lots of Notolab noise)
h) .02
Relative Increase in Avg Code Compression:
a) .03
b) .3 (big difference)
c) .05
d) .01
e) .08
f) .01 (We did manage to cause fewer things to be pushed to the stack in some cases, but larger functions that used callee-saved registers seemed to still use them anyways, so no major impact here)
g) .03
h) .02

Optimization Passes : Strength Reduction To implement strength reduction, we inspected the gcc output for various divide and modulus instructions that were present in the benchmarks and we copied the magic number formulas as preset "code snippets" to be output by our x86 generator when the appropriate operation is detected. This massively improves speed, but also seriously hurt code size since a single divide turned into a series of 5 or 6 instructions. This affect was negligible until inlining was implemented, because our inlining heuristic counted the number of lines before the magic number formula was elaborated, and so inlined functions ended up being much larger in reality than our compiler intended, meaning large divide-heavy functions were copied extensively through our code. We intend to remedy this by weighting divides differently than other instructions in the inlining process.

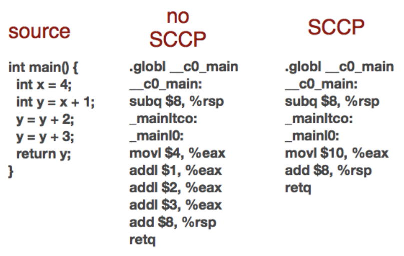
Performance Evaluation
Git Commit : 184097eb7a27441559bea3812f9abd80c9792fef
Lab5Benchmark Submission ID: 52
Relative Increase in Avg Benchmark Speed: .04
Relative Increase in Avg Code Compression: -0.02

Optimization Passes : Register Coalescing We implemented greedy register coalescing. Throughout register allocation, we keep track of temporaries that can be coalesced with one another with "alias" edges that are added when we detect a move instruction. We remove these "alias" edges if either temporary is assigned to again later in the program (while the other is still live), since this would break the assumption that they can be places in the same register. Otherwise our algorithm is the same as presented in lecture [2]. During greedy coloring, we combine the constraints for all coalescable temporaries and assign them all the same color. While this is not necessarily optimal in all cases (specifically the case where the extra constraints require an additional register to color the graph), it did, on average, make other optimizations such as SCCP and PRE drastically more efficient (since they both introduced many extra temp -¿ temp moves that were prime suspects for colaescing and mitigated their code-size impact). We intended to only coalesce if it did not require spilling, but that additional logic seemed to not affect the score, and so we left it out.

Performance Evaluation
Git Commit : 8e5bfca99021afd5205181f0925434a6301d0f51
Lab5Benchmark Submission ID: 7
Relative Increase in Avg Benchmark Speed: .05
Relative Increase in Avg Code Compression: 0.01

Optimization Passes : SCCP The SCCP algorithm was implemented using the pseudocode provided in the Cooper textbook [3]. We found that implementing SCCP had a larger impact on our code size than on performance. At a high level, the algorithm worked by maintain a "lattice" for each variable in each definition and use with information pertaining to whether there was evidence a variable was assigned to, if it was assigned a constant, or if it could have multiple values known only during runtime. By operating on the CFG and SSA graphs, the algorithm iteratively propogated lattice values for each variable throughout the program, until convergence. Edges on the CFG that were never visited (in the case where a value inside the conditional was known), were deleted. If a CFG node no longer had any parents by the end of the algorithm, it was deleted. Specifically, after implementing SCCP (around submission 55), we saw an approximately 5% increase in speedup, but about a 25% code size reduction; this was primarily a result of eliminating conditional branches and basic blocks, allowing us to remove large chunks of dead code from the final output. Given that most of this code was unreachable, the performance benefits of SCCP came from propagating constants throughout the program and reducing the overall number of computations — this is especially apparent when running our compiler on L1 programs, as most are condensed into a few short lines of assembly. After running Dead Code Elimination and SCCP on the tests/bench/jen.l4, we were able to reduce the output file size by over 2500 lines of code, a substantial improvement from our lab 4 compiler. The psudeocode for SCCP can be found on pages 577-579 of the Appel textbook [7].

The following diagrams show a simple L1 program before and after SCCP:



```
int main() {
  int x = 4;
  int y = x + 1;
  y = y + 2;
  y = y + 3;
  return y;
}
```

```
.globl __c0_main
__c0_main:
subq $8, %rsp
_mainltco:
_mainl0:
movl $4, %eax
addl $1, %eax
addl $2, %eax
addl $3, %eax
add $8, %rsp
retq
```

```
.globl __c0_main
__c0_main:
subq $8, %rsp
_mainltco:
_mainl0:
movl $10, %eax
add $8, %rsp
retq
```

Fig. 4. Comparison of SCCP assembly output

While this program is small, the effects on larger programs are much more apparent, with the code size being reduced to only a few lines for most straight line code. Results are shown below, run on the Notolab servers.

Performance Evaluation
Git Commit : 2bae3d40aa9172049e55568f1bc362ac15b8360e
Lab5Benchmark Submission ID: 61
Relative Increase in Avg Benchmark Speed: .05
Relative Increase in Avg Code Compression: 0.04

Optimization Passes : PRE We implemented the Partial Redundancy Elimination algorithm from lecture [4], utilizing our now-generic dataflow framework to perform the 4 dataflow passes. The only modification we made to the lecture algorithm (correction: the only *intended* modification) was that we explicitly disallow reordering of divide and mod and shift expressions if we are on safe mode, since these could cause an exception. We achieved moderate results with this algorithm for the following reasons: a) The algorithm as implemented finds the 'latest-but-useful' placement of any global expression. This is an effective definition for top-down analysis, but it does not encompass loop-hoisting. It is possible we still have a bug but we agree with out implementation after analysis of the following sources [5]. b) The Dataflow passes themselves were quite costly in terms of compiler runtime, and so we only run it on -O1 c) Out of concern that SSA (since it renamed temporaries until very late in our development cycle) would cause PRE to miss potential hoisting locations by obfuscating equivalent expression computations, we always perform PRE before our SSA module. This meant that for large cases, it was operating on code before it had been properly pruned of dead statements and expressions. d) in order to get loop hoisting *somewhat* functional, we inserted loop preheaders unconditionally, which helped provide a location for PRE to place expression computations but left us with redundant labels Also of note is that Coalescing proved crucial to PRE, since PRE replaces expressions with new temporaries, many of which can be coalesced. This alone brought the average speedup of PRE from about .5 to the .9 shown below. The .5 metric was evaluated informally, since we implemented coalescing first. Regardless of its benefit, we found the effort quite exciting and fruitful, and our results as evaluated by Notolab are shown below. Of note is the extremely significant increase in code size. This is due to both the added labels for loop preheaders (which we found inherently necessary for PRE to have a place to put expressions), and the fact that for expressions without redundancies, our PRE algorithm still inserts a termporary, introducing an unecessary move that would be removed by constant propogation if we had it.

Performance Evaluation
Git Commit : d4e35de7a69ceadccf4f7e2d86c1d9bcacf0b4e5
Lab5Benchmark Submission ID: 14
Relative Increase in Avg Benchmark Speed: .09
Relative Increase in Avg Code Compression: -1.4

Optimization Passes : DCE We forwent implementing Aggressive Dead Code Elimination (ADCE) in favor of Dead Code Elimination (DCE) for two primary reasons. First, the

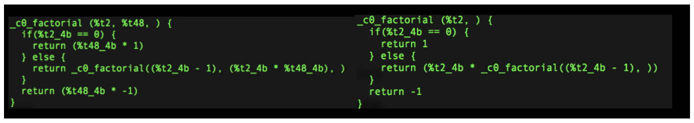
DCE algorithm was far simpler to implement than ADCE, thereby giving us similar performance with far less effort. Second, the amount of literature available on DCE was substantially greater than that for ADCE [7], allowing us to gain a better comprehension of the algorithm. The DCE algorithm is based on the psuedo code in Appel textbook. Specifically, it looked at each assignment statement, and determined if a variable defined by a statement had any uses in the program. If it didn't, it would eliminate the line of code. We had to be careful not to eliminate any code with side effects. In particular, returns, function calls, memory assignments, division by 0, and modulo by 0 were assumed to have side effects. Therefore, any statements with these operations were "protected" from being deleted, unless unsafe mode was on. After implementing DCE, we saw both our performance increase and code size decrease. Specifically, we implemented DCE around submission 45, and immediately saw a 4-5% increase in performance. At the same time, we were able to trim our code size by about 3-4%. These figures are backed up by the Notolab results shown below.
The psudeocode for DCE is reproduced below:

```
$W \leftarrow$ a list of all variables in the SSA program
while $W$ is not empty
    remove some variable $v$ from $W$
    if $v$'s list of uses is empty
      let $S$ be $v$'s statement of definition
      if $S$ has no side effects other than the assignment to $v$
        delete $S$ from the program
        for each variable $x_i$ used by $S$
            delete $S$ from the list of uses of $x_i$
            $W \leftarrow W \cup \{x_i\}$
```

Performance Evaluation
Git Commit : 11b3ae73927e5b51525739c4681ef8d8b170d510
Lab5Benchmark Submission ID: 50 (earlier was 47, but that broke l4-large)
Relative Increase in Avg Benchmark Speed: 0.04
Relative Increase in Avg Code Compression: .028

Optimization Passes : Function Inlining Function Inlining did not seem to have a massive positive effect on runtime. In order to implement Inlining in the codegen stage, we simply copy the contents of the function in question into the target program, renaming all temporaries and labels as we go (this is necessary to avoid collisions if you inline the same function more than once). We then make sure the result is stored in the proper destination, and the rest seamlessly integrated itself with downstream stages of our compiler. To avoid having to do checks for mutually-recursive functions, we decided to disallow inlining for any functions that call other functions. We believe we are justified in this because the target program still has to set up a callframe, and so these are not the small functions we intended to inline. Regardless, we used a simple line-length metric to determine whether a function was inline-able or not. This we have analyzed to be the root-cause of its supposed inefficacy. While this is not a bad metric, a large consideration for inlining functions is the loop depth of the call which is not considered. Also explored was the hit to code size we experienced once magic number divide was implemented, as stated in the earlier strength-reduction section.
Performance Evaluation
Git Commit : 8161514d9d8b07274673095964f22dd259632ec3
Lab5Benchmark Submission ID: 42
Relative Increase in Avg Benchmark Speed: 0.01 (polluted by local variation/notolab noise)
Relative Increase in Avg Code Compression: -.03

Optimization Passes : TCO One of our more fruitful optimizations was TCO. We do more advanced TCO than the naive solution (which is to simply detect recursive calls and branch back to the beginning of the stackframe), and actually permute the source code of the file (in the translation stage) to accept extra arguments for multiply or add accumulators if they are detected. This significantly sped up code such as the small snippet below:



```
_c0_factorial (%t2, %t48, ) {
  if(%t2_4b == 0) {
    return (%t48_4b * 1)
  } else {
    return _c0_factorial((%t2_4b - 1), (%t2_4b * %t48_4b), )
  }
  return (%t48_4b * -1)
}
```

```
_c0_factorial (%t2, ) {
  if(%t2_4b == 0) {
    return 1
  } else {
    return (%t2_4b * _c0_factorial((%t2_4b - 1), ))
  }
  return -1
}
```

Fig. 5. Comparison of TCO modified source vs Original

And benchmarks like monica and mist benefited greatly on runtime from this optimization. Also of note however, is a minor hit on code size. This is because we added an additional argument to the function, and so now all callsites became more complex in the fact that they had an additional argument to load.
Performance Evaluation
Git Commit : 5f9803a1d7950f6255a9296c55ea08752be315dc
Lab5Benchmark Submission ID: 48
Relative Increase in Avg Benchmark Speed: 0.04
Relative Increase in Avg Code Compression: -.001

Optimization Levels Our compiler operates on two optimization levels: O0 and O1. The performance of our compiler varies significantly between the two:
In O0 (also default), we forgo expensive optimizations, only favoring those that run relatively quickly (comparable to other required parts of the compiler). Specifically, we run:

- Tail Call Optimization
- Function Inlining
- Coalescing
- Magic Number Division and Modulo
- Basic Block Reordering
- Constant evaluation
- All other Lab3 and 4 simplification passes

In O1, our compiler takes more time to run; however the code produced is much smaller in size and more performant. In addition to the optimization in O0, we also run:

- PRE
- SCCP
- DCE

and it is important to note that these three require setup in the form of SSA and CFG critical edge splitting. In Unsafe mode (default is safe), the following effects are felt:

- Null checks on pointers, shifting checks on Lsh and Rsh, Array Bounds checking and Array Allocation checking (positive size) are all discarded.
- We allow divide/mod instructions to be reordered by PRE
- We allow divide/mod instructions to be eliminated by SCCP and DCE

Interaction Between Optimizations

While many cases of this has already been explored in the paper above, we reiterate the most interesting interactions below for your convenience.

Analysis Passes

- CFG Simplification versus Critical Edge Splitting: Splitting critical edges undoes much of the compression work that CFG simplification achieves. Therefore we ended up having to bookend major modules like PRE and SSA with Critical-Edge Splitting before and CFG Simplification Afterwards.

Optimization Passes

- PRE and SSA: These two interacted quite heavily at first, when we still renamed temporaries when transitioning out of SSA. Because of this renaming, we ended up aliasing expressions that would otherwise be reordered by PRE. Therefore we run PRE first in the processing chain.
- Inlining and Strength Reduction: On average on the test cases, we saw inlining lower our code size score approximately 0.01. With Strength Reduction this drastically increased (threefold) due to the order in which we performed them. Since the line length for function inlining was evaluated at the abstract code-triple stage, relatively short functions were inlined that contained many divides and mods. However when these reached x86 codegen, the magic gcc formulas blew up the code size, meaning we ended up inlining many instances of large functions. In our next compiler we would attempt to weight each instruction according to how large we expect the x86 to be.

Source Code Overview : in order of execution

- Top.ml - the top level module that contains the main compile function
- C0_lexer.mll/C0_parser.mly - Lexer/Parser
- funtable.ml/mli - Global Declaration Typechecker (typecheck global struct definitions, file order, function declarations, etc. . .)
- typechecker.ml/mli - Typechecker
- trans.ml/mli - Translate and Elaborate. Also TCO manipulations
- codegen.ml/mli - Generate Code Triples. Also Inline
- cfg.ml/mli - Create CFG
- pre.ml/mli - Run PRE (with 4 dataflow passes), output CFG
- dom.ml/mli - Create Dominator Trees for SSA, output a dominator tree
- ssa.ml/mli - Transition into SSA form with the CFG and the Dom Tree, run DCE, run SCCP, transition out of SSA form
- liveness.ml/mli - Performs liveness analysis and flattens the CFG
- regalloc.ml/mli - Perform register allocation and coalescing
- xcodegen.ml/mli - Convert code triples to x86 code and print to file. Gcc Magic Divide code snippets are substituted for divide and mod.

Performance Metrics. The following graph displays the approximate proportional runtime of each module. We arrived at these numbers by profiling our code during testing regression from L1 to L4.

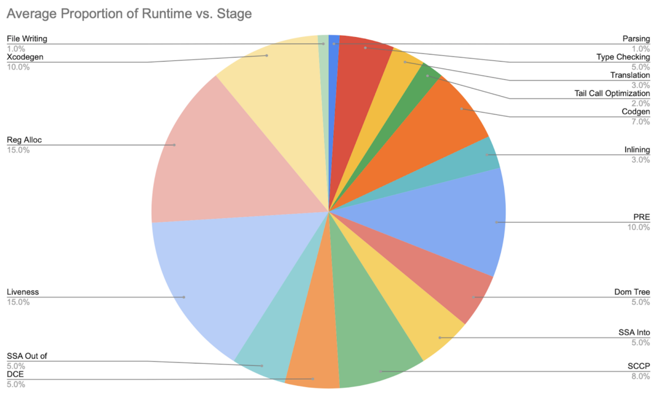


Fig. 6. Proportion of Compiler Runtime Per-stage

The average speedup and code size of our compiled programs vs submission number is shown below. As we implemented more optimizations, our compiled program became more performant and the executable became smaller. The approximate submission we successfully implemented each optimization is annotated on the graph below. The data was obtained from our Lab 5 Notolab Benchmark submissions.

Of note are the major dips in the graph. Aside from random Notolab noise, these each signify a point where optimizations we intended to compliment eachother ended up destructively interfering. Examples of this are the dip after 'Strength Reductions' where our larger divide code negatively impacted our inlining and the dip after SSA where the extra moves ended up severely inhibiting the ability of our register allocator and colaescing module to limit stackslots (the temporary chains just got too large, and we only check first-degree colaesce neighbors in our module, never transitive neighbors). Also of note is that we consistently traded code size for speed almost all the way to the end, until we implemented optimizations like DCE and SCCP that started to help both.

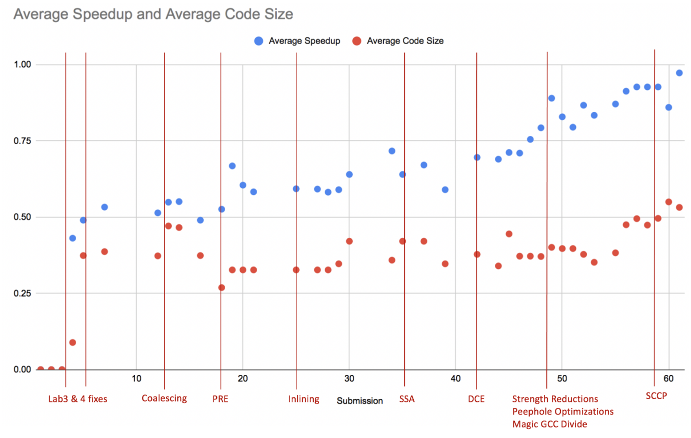
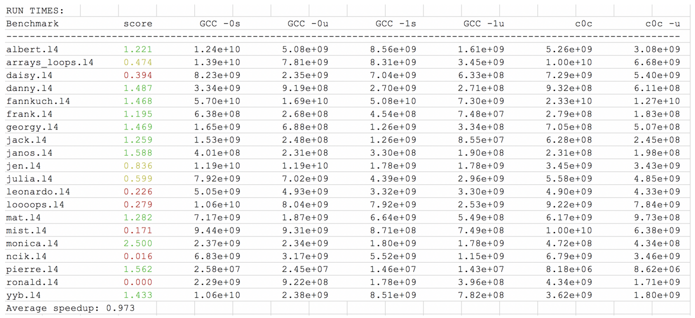


Fig. 7. Compiler Benchmark Speedup and Code-Size over time, with relative development milestones

Aside from the interactions explicitly mentioned, most other optimizations were purely additive. This is why we are confident in our performance evaluation section above, where we attempt to isolate the gains on an opt-by-opt basis/

Time vs. Space Tradeoff While we have mentioned this already, and shown several optimizations that add to performance and detract from code size reduction, we reiterate here the basic takeaways for us. a) Most optimizations are a tradeoff between these metrics, the only win-win optimizations we see are when we combine optimizations (such as SSA WITH Coalescing to counteract the extra moves inserted). b) When trading between these metrics, we almost always chose speed. We implemented DCE primarily to counteract the fact that many of our more performant optimizations hurt code size.

Analysis of Compiler performance on Benchmarks To ease this analysis, we have provided the highest-performing run statistics below on the benchmarks. We feel the provided benchmarking quite adequately highlight the benefits and pitfalls of using our compiler:



RUN TIMES:

Benchmark	score	GCC -0s	GCC -0u	GCC -1s	GCC -1u	c0c	c0c -u
albert.l4	1.221	1.24e+10	5.08e+09	8.56e+09	1.61e+09	5.26e+09	3.08e+09
arrays_loops.l4	0.474	1.39e+10	7.81e+09	8.31e+09	3.45e+09	1.00e+10	6.68e+09
daisy.l4	0.394	8.23e+09	2.35e+09	7.04e+09	6.33e+08	7.29e+09	5.40e+09
danny.l4	1.487	3.34e+09	9.19e+08	2.70e+09	2.71e+08	9.32e+08	6.11e+08
fannkuch.l4	1.468	5.70e+10	1.69e+10	5.08e+10	7.30e+09	2.33e+10	1.27e+10
frank.l4	1.195	6.38e+08	2.68e+08	4.54e+08	7.48e+07	2.79e+08	1.83e+08
georgy.l4	1.469	1.65e+09	6.88e+08	1.26e+09	3.34e+08	7.05e+08	5.07e+08
jack.l4	1.259	1.53e+09	2.48e+08	1.26e+09	8.55e+07	6.28e+08	2.45e+08
janos.l4	1.588	4.01e+08	2.31e+08	3.30e+08	1.90e+08	2.31e+08	1.98e+08
jen.l4	0.836	1.19e+10	1.19e+10	1.78e+09	1.78e+09	3.45e+09	3.43e+09
julia.l4	0.599	7.92e+09	7.02e+09	4.39e+09	2.96e+09	5.58e+09	4.85e+09
leonardo.l4	0.226	5.05e+09	4.93e+09	3.32e+09	3.30e+09	4.90e+09	4.33e+09
loooops.l4	0.279	1.06e+10	8.04e+09	7.92e+09	2.53e+09	9.22e+09	7.84e+09
mat.l4	1.282	7.17e+09	1.87e+09	6.64e+09	5.49e+08	6.17e+09	9.73e+08
mist.l4	0.171	9.44e+09	9.31e+09	8.71e+08	7.49e+08	1.00e+10	6.38e+09
monica.l4	2.500	2.37e+09	2.34e+09	1.80e+09	1.78e+09	4.72e+08	4.34e+08
ncik.l4	0.016	6.83e+09	3.17e+09	5.52e+09	1.15e+09	6.79e+09	3.46e+09
pierre.l4	1.562	2.58e+07	2.45e+07	1.46e+07	1.43e+07	8.18e+06	8.62e+06
ronald.l4	0.000	2.29e+09	9.22e+08	1.78e+09	3.96e+08	4.34e+09	1.71e+09
yyb.l4	1.433	1.06e+10	2.38e+09	8.51e+09	7.82e+08	3.62e+09	1.80e+09
Average speedup: 0.973							

Fig. 8. GCC Comparison Results

Discarding the runs with safety checks in our analysis:
We perform very well on programs that have

- TCO-able function calls
- Lots of dead code (that is not in a loop)
- Radically inefficient control flow - our CFG simplification techniques are quite sophisticated
- Lots of branching on comparisons - we optimize this nicely
- Code that can be simplified with constant prop
- Variables that are prime for coalescing

We perform very poorly on programs that have

- Dead code in loops (because do not have rigorous frameworks for detecting loop-invariant expressions. After we theoretically hoist expressions, our existing code could eliminate it) *example benchmark julia.l4 lines 61-70*
- Loop-invariant computation that is expensive (because we do not have Loop-Invariant Code Motion) *example benchmark arrays_and_loops.l4 lines 30-38*
- Nested struct accesses (because the way we handle struct dereference operators in codegen is inefficient in the sense that we do not condense the offsets into a single instruction, and require that SCCP take the full brunt of this simplification. It also may not be apparent at that stage that nested struct offsets can be combined because they can be aliased with temporaries). *example benchmark daisy.l4 lines 151-152*

Summary and Future Work Thank you to the course staff for enabling such an ambitious project as this one. We both had a lot of fun implementing it. Our plan, given more time to bring our score up, would be as follows: Implement Loop-Invariant Code Motion, Implement Alias Analysis, Perform Loop Tiling

Github Submission Link https://github.com/15-411-F20/Interligua/tree/2bae3d40aa9172049e55568f1bc362ac15b8360e

REFERENCES

[1] http://www.cs.cmu.edu/afs/cs/academic/class/15411-f20/www/lec/06-ssa-into-outof.pdf
[2] http://www.cs.cmu.edu/afs/cs/academic/class/15411-f20/www/lec/02-registerallocation.pdf
[3] Engineering A Compiler, 2nd edition, Keith Cooper et al.
[4] http://www.cs.cmu.edu/afs/cs/academic/class/15411-f20/www/lec/17-pre.pdf
[5] http://rsim.cs.uiuc.edu/arch/qual_papers/compilers/knoop92.pdf
[6] Optimization in Static Single Assignment Form, December 2017, Tokyo Institute of Technology
[7] A. W. Appel, Modern Compiler Implementation in ML, Cambridge: Cambridge University Press, 1997