Project 5 Design Document: Compiler Optimization

DeJuan Anderson (dejuan@mit.edu), Maddie Mott (mmott@mit.edu), Youyang Gu (yygu@mit.edu)

- Division of labor
    - Maddie: CSE(actual working version), debugging of Register Allocation for codegen, General Debugging Assistance
    - DeJuan: CSE (draft), DCE, Constant Folding, Assembly hacks/modifications, Design Document
    - Youyang: Register Allocation (generating reaching definitions, counting spill costs, creating interference graph, k-coloring of nodes, assigning registers to Var)

What to write about:

The writeup for this project is extremely important. Although it explicitly accounts for only 20% of the grade, it will also be used to determine your score for the Implementation aspect of the project (40%).

Your written documentation must discuss each optimization that you considered. The thoroughness of your exploration of the optimization space will be an important aspect of your grade. For each optimization that you implement, your writeup must convince the reader that the optimization was beneficial, general, and correct. For each optimization that you decide not to implement, you must convince the reader that the optimization would not be beneficial given your generated code for the given benchmarks and our target architecture.

At the outset of our optimization phase, we had a meeting to discuss the possible optimizations we could implement, and which ones we would do and in what order. We considered CSE, DCE, register allocation, constant folding, constant propagation, copy propagation, right shift replacement for division, assembly code manipulation, algebraic identity simplifications, loop invariant code motion, unreachable code elimination, list scheduling, instruction selection, and data parallelization. After this brainstorm about the possible optimizations, then after coming up with a list, assigned a priority to each one.

You may recall that at the outset of Project 5, our CSE was not properly working for all cases. We resolved that fixing that was of high importance, as CSE is both very useful at avoiding recomputation of expressions and we already had an implementation at hand that needed fixing, rather than implementation from scratch.

Considering that CSE creates compiler generated temporary variables for *every* expression and then uses only the ones that can be safely replaced, for CSE to not slow down the program, DCE was a natural solution and became the next optimization we decided to

implement. DCE would be able to remove all the compiler generated temps that were not actually used to do replacement and clean up other unused pieces of code at the same time.

After thinking about these two optimizations, we considered the rest and assigned priorities based on our decision of how hard an optimization would be to implement in our code representation model versus its projected benefit.

Since regardless of architecture, memory accesses are far slower than register manipulation, Register Allocation quickly rose to the top of the list when we considered possible benefits, and it was decided that we would definitely implement it as well.

We came up with four levels of priority:

1) Optimizations in this level either were already near implementation, were essential for other optimizations, or would offer incredible speedup despite their difficulty. The optimizations we assigned to this category were DCE, CSE, and Register Allocation.

2) Optimizations in this level had no framework laid for their implementation and were very difficult, but would offer significant speedup. We would do these after finishing the optimizations in the previous priority level. Optimizations assigned to this priority were refactoring CSE to not use global temporary variables, constant folding, algebraic identity operations and right shifts, loop invariant code motion, and assembly manipulation.

3) Optimizations in this category were projected to be more difficult to implement than the potential payoff, or easy to implement but with low payoff, and so were left until we had completed all optimizations in both priority level 1 and 2. Optimizations in this priority level were constant propagation, other arithmetic optimizations like strength reduction, copy propagation, and unreachable code elimination.

4) Optimizations in this final category were deemed unimportant for implementation due to either being incredibly complex, or being automatically handled by the hardware in the machine. Optimizations left here were list scheduling, instruction selection, and data parallelization.

I have already discussed the optimizations in priority level 1, and will now cover each of the optimizations in the remaining priority levels.

For the first optimization in priority level 2, refactoring CSE to not use global temporaries, we noticed that in trying to get CSE to work for Project 4, a simplifying assumption that greatly affected speed but made CSE easier to implement at the time was to allow all compiler generated temps to be globals instead of keeping track of their scope. Because of this compromise, our CSE implementation was generally actually causing slowdown compared to

the unoptimized version. Making the temporaries behave as normal variables would fix this issue, and CSE was already in debugging phases, so this was an important change to make.

As for constant folding, due to some of our architecture, namely SPSets, we could implement those changes with relative ease automatically as part of CSE, so we ranked it high due to ease of implementation. For loop invariant code motion, since loops are hot regions in the code that are frequently executed, we thought that performing loop invariant code motion could greatly speed up the execution of loops. Unfortunately, due to the constraints of our code representation, actually doing the code hoisting and/or creating new locations to hold the relocated data was found to be non-trivial, and in terms of time required versus other potentially implementable optimizations, regrettably this was not implented by the due date of this project.

Lastly,for assembly manipulation, we realized that a lot of our assembly code involved useless sequences of instructions like push x pop x (where x is some memory location) or an unconditional jump to a label directly beneath the jump, making the jump completely unnecessary. We also realized that due to having built our compiler on a stack based model, there were many instructions like push x pop y to move a value from one place to another when we could have simply done movq x y instead. Hacking directly on the assembly instructions to remove these extraneous commands and perform the transformation push x pop y into movq x y would greatly increase efficiency by directly reducing the number of instructions while making some of the remaining instructions more efficient.

Unfortunately, due to time constraints and unforeseen difficulties in getting optimizations to work with our generated code structure, we were unable to implement any optimizations at level 3 or lower; however, as these are in the lower half of priorities, we believe that the lack of their inclusion, while lamentable, is not a tragic loss that kills our hopes for efficiency.

You should include assembly or IR code examples for each optimization. Show how your generated code is transformed by the optimization (might be hand-applied for optimizations you did not implement). Highlight the benefit of the optimization on the assembly or IR code. Discuss exactly how the benefit was achieved given the characteristics of our target architecture. Include empirical evidence that proves your conclusion for the optimization.

Now that we have provided the reasoning behind which optimizations we have implemented, we will cover how the optimizations actually work, trace through what each optimization does and explain why it is correct.

Let us begin with first explaining our representation of the code at the level where our optimizations work. We discretize the code into a graph structure, where the nodes in this graph structure are “FlowNodes”, an interface we created to represent code in a graphically meaningful way. The FlowNode abstract class has five implementations: Codeblock, Branch, NoOp, START, and END.

Codeblocks represent standard sequences of instructions, such as Declarations, MethodCallStatements, and Assignments.
Branches represent points where a condition is used to direct control flow down one of multiple possible paths in a forward analysis; in other words, it represents points where we have if(), for(), or while() statements.

NoOps represent join points for a forward analysis, and split points for a reverse analysis. They are basically the opposite of Branches. They are a convenient point at which to join multiple control flow paths without worrying about execution of code at the join point. For if() branches, the NoOp is the point at which the two different branches meet (if they do). In the case of for() and while() loops, the NoOp is the false branch of the loop's condition, since if controlflow advances linearly through the program that is the first point any path through the top of the loop must encounter.

STARTs represent either the start of a method, where the parameters for that method are officially recognized, or the start of an inner loop. When initializing an inner loop, Codeblocks initializing, assigning, and incrementing where the loop-controlling induction variable are inserted as necessary.

ENDs represent a point where control leaves the current method, and hold the data for return statements, if one exists.

We scan the decaf program and construct from our IR a graph using these five node types to represent the program in a semantically meaningful way, linking blocks with parent and child pointers if control flow passes from one to the other, and creating one graph for each method in the program. The graphs for different methods in the same program are therefore disjoint; we can easily pick which method we wish to analyzing by choosing its respective START node. We can walk the graph beginning from this START and following child pointers to conduct forward analyses, or scan for the ENDs and walk backward along the parent pointers to conduct backward analyses.

Now that we have covered the structure of our representation, we can describe how the optimizations work.

As for DCE, we first need to compute liveness. To assist in this computation, we construct a Bitvector, a class which directly implements the idea of bitvectors as discussed in lecture; each variable has an entry, with a 0 or 1 to indicate whether or not it is alive. Bitvectors can be unisoned and compared; the unison will take any slot that is 1 in any of the bitvectors being unified, and ensure that the resulting unification will have a 1 in that slot. Only slots which are 0 in all bitvectors being unified remain zero after the unification.

For each method, we create a bitvector with an entry for every variable used in the program. After this, we execute a forward scan, working from the START down to the last END in the program, adding all ENDs we encounter into a list. As we do this walk, for every node we encounter, we assign a bitvector with all zero entries to each FlowNode we encounter. Once completed, we choose an END to start from.Before I continue, let me define IN and OUT for a given FlowNode; since we are working in reverse, IN implies we are coming from a child of the node into that node, while OUT implies we are leaving the node to visit one of its parents. Therefore, the IN for this final END is an all zero bitvector.

After computing the IN for the END node, we flip all entries for global variables, method arguments, and any variables used in the return statement (if there is one) to 1, and proceed from the END node, following parent pointers. This newly edited Bitvector is the OUT of the END. For each node we visit, we take the unison of all the child node bitvectors. For nodes whose children we haven't visited, this will be a unison with all zero bitvectors which is an identity operation. The only time this occurs is in branches or loops. However, by following the parent pointers from the branch condition which starts the loop, we will eventually come back to this loop from the unvisited child, ensuring that we will see every node.
We apply a fixed point algorithm; until the INs and OUTs of all bitvectors in the method converge, we continue to loop through the pointers. Since we take unisons at every step, the amount of information never decreases, so we are guaranteed to eventually converge. Once we have converged, we start iteration from any other ENDs in the same method, to ensure that we converge to a solution which incorporates all control flow paths. After computing this fully convergent model, we then simply search out the codeblocks, and for each codeblock, walk in reverse up the list of statements contained within. If we encounter a dead assignment, we remove it.

An example:

Codeblock:
x = 5;
y = 42;
y = 69*z*y;
x = y/2;
z = x;

Assume the bitvector, with positions XYZ, comes in with value 001, meaning only Z is alive. As we walk up this codeblock, we find an assignment to Z; we check the bitvector. Z is alive, so look at the right hand side. Flip X's bitvector to 1 to indicate liveness due to usage in live assignment. The right hand side does not expose an upwards use of z, so flip Z's bitvector to zero. We now have the bitvector 100 after processing z = x. For x = y/2, we get 010.
Now, for y = 69*z*y, we note that z in in the right hand side, so we flip its bitvector. However, since y is also in the right hand side and this assignment was alive, this assignment exposes an upper definition of y, so we do not flip its bitvector to zero, giving us a result of 011. Then

we see y = 42. Y is still alive because of the upwards exposed definition below, so only now do we flip y to zero : 001. Lastly, we find x = 5. X's bitvector is 0, so we remove the statement entirely.

This implementation is correct because it fully calculates the final liveness model before the actual optimization is applied, and preserves definitions in the case when a previous assingment exposes an upwards use. There is handling for method calls as well; all things used in a method call have their bitvector flipped to 1.

For CSE, we realized we would need a way of recognizing equivalent expressions, such as (a * c * b) and (a * b * c). The method we chose to solve this problem was the creation of SPSets which have an operator, showing which operation an SPSet represents, and structures for holding each of the various components of any expression (SPSets representing nested operations, Vars, IntLits, BoolLits, MethodCalls, Ternarys, and Comparisons). It was necessary to represent ternaries and comparisons with their own structures (SPTern and SPComp respectively) because unlike the other operations, they are not naively commutative and associative. We then implemented contains(), remove(), and replace() methods for each of SPSet, SPTern, and SPComp, so that they can easily determine if an Expression is a subexpression of themselves, and substitute a variable for a subexpression as needed. To allow SPSets to be used easily as keys into HashMaps in the actual CSE, we then wrote equals() methods sch that two SPSets would be equal iff they represented the same expression. To solve the problem of variables being reassigned, SPSets contain ValueIDs representing variables, where CSE must maintain a mapping where any ValueID will always map to exactly one value, and ValueIDs can be converted to the correct variable at any point in the program.

For example, the expression 3 + 4 * 5 + (8 - 3) would be represented by
SPSet (+, 3, SPSet(*, 4, 5), SPSet(+, 8, Neg(3)), so we could correctly determine that -3 + 8 is a subexpression with the appropriate recursive calls.

Having made a representation for expressions which can handle commutativity and checking for subexpression, we then made MapContainer, which tracks expression availability to maintain a bidirectional mapping between unique ValueIDs and variables. When we wish to check an expression for common subexpressions, we can see if its SPSet contains any of the SPSets currently stored in variables available at this point in the function, and substitute them as appropriate. In order to actually track availability, each FlowNode is mapped to a MapContainer, and the variables available at a FlowNode is the intersection of the sets of variables available at its parents. When we assign a variable, its associated ValueID changes to the new value. Because all assignments are also stored in a compiler-generated temp (unless that assignment is eliminated by DCE), the original value of the expression stays available. Because we cannot evaluate expressions at this stage in the program, assignments to array elements are treated as though any index of the array could have been

the one assigned, and they all are considered to have no known ValueID. To uniquely identify variables, we use IR_FieldDecls, since there is one IR_FieldDecl for each variable.

If at any expression we do not known what the value of any of the variables used in it is because of ambiguity, we choose not to apply CSE to that expression. For example:

```
d = a + b;
e = b + c;
if ( a == b) {
   c =  1 + 3;
} else {
   c = 5;
}
bad_luck = a + b + c;
```

We don't know the value of c, since it could be one of two different ValueIds, and so cannot fully parse the expression into an SPSet. We therefore will choose not to apply CSE to the assignment to bad_luck, even though it is in theory eligible.

Unfortunately, the way we implemented the intersection logic for available expressions as the intersection of all parents means that passage through a for or while loop makes all expressions unavailable, since we cannot calculate the expressions available at the beginning of the loop without knowing the expressions available at the end of the loop and vice versa, so we make the conservative assumption that no expressions are available. This is suboptimal for program efficiency, but is guaranteed to be correct.

Register allocation was done in 4 parts:

- Computing spill cost for each FieldDecl: a similar heuristic to the one in class is used (where loops are favored). This is done by going through the code and keeping track of how many loops we are in.
- Generation of Reaching Definitions (using the algorithm described in lecture). It is then used to make Webs. Webs are made such that all assignments of a fieldDecl that need to be in the same register are placed in the same web.
- Construction of the Interference Graph, where each web is assigned to a GraphNode, and an edge is created if two webs overlaps.
- Coloring: The GraphNodes are 4-colored using the algorithm as described in lecture. Registers are assigned to each GraphNode. Spilled nodes are computed using spill cost.
- The register assignments are placed in the logic for codegen. If a register exists, codegen (AssignRegisters.java) will use that register to locate/assign the variable. Otherwise, it will use the stack.

For constant folding, as was mentioned in CSE, we have a representation for equations known as an SPSet. Because this representation disassociates order from equations and

stores everything related to a common operator in lists which contain only certain types of equation components, by simply looking at the operator and the int list or boolean list, we can easily compress operations. For example; (+ {3, 5, 7, 9}) represents 3+5+7+9, and we can easily do constant folding by simply iterating over the integer list, and summing its members, (+{24}).

For booleans, we can compress the list based on the operation. For example, say the operation is &&, and we have (&&{true, true, false}, (method call), (||{other things})) Though at first this seems incredibly complicated, if we look at the first entry, we see that we have true && true && false. This simplifies directly to false, so we have false && method call && other complicated things. We do not care what the method call resolves to, nor do we care what the other things are; since the SPSets maintain unordered sets of all things united by a common operation, having resolved the plain booleans to false allows us to resolve the entire expression to false, so the constant folding skips straight to ({false}) at this point.

For the assembly hacks and modifications, we look at the fully generated list of assembly. I will give a series of examples. In this example, for the sake of simplicity, any single letter(i.e. x, y, a, b, etc) represents some sort of memory location, whether it be an offset from the base pointer, a register, or a literal.

We have implemented four different assembly modifications:
push x pop x → deleted;
push x pop y → transform [push x] into [movq x y], delete [pop y]
jmp L10 immediately followed by L10 next line → remove the jump
add or subtract 0 x → delete the add or subtaction

For the last optimization, correctness is easy to see; simply removing x + 0 or x - 0 is getting rid of an instruction that did nothing.

For push x pop x, note that, if I were to convert the instructions into English, this would be “Put the value at location x on the top of the stack. Now, pop the value on the top of the stack into location x.” This instruction series puts a value on the stack and immediately moves it back to where it was; there is absolutely no need to do that and doing so slows down the program, so removal of this is also correct.

For push x pop y shifting to movq x y, note that what we are doing is “Put the value at location x onto the stack. Now move the value from the top of the stack into location y.” Instead of doing a memory store then load, why not simply move the value from x to y? This reasoning is why we made the change, and from that perspective, we can see why it would be correct; the exact same action is accomplished, just in a far more efficient manner.

Lastly, as for the jump removal; in that situation, we have code that says “jump to this location in memory” where the location indicated is quite literally the next line in the assembly. In that

case, we shouldn't need to waste time doing the jump at all, and removing it allows us to fall directly into the location desired without the overhead of a meaningless jump.

Earlier I mentioned that these were heavily optimized so that they can work in a single pass. The way this is done is, for any time we detect an optimization can be made, we begin a search. In this search, we check the instruction we just processed and two instructions ahead; if those instructions match one of the optimizations, apply it, and check again on the instructions outside of those.

So for example:
push x
push z
pop z
push y
pop y
pop y

We move top to bottom. push x push z doesn't trigger anything. However, when we hit push z, we notice that the next instruction is pop z, and we mark both for deletion.
push x <--outside opt
push z DELETE
pop z DELETE
push y <--outside opt
pop y
pop y

Next we check if push x push y matches an optimization pattern, since those are the two immediately bordering our optimization. It doesn't, so continue.
We skip pop z since it's marked for deletion.
Now we step forward and see push y, and notice the next instruction is pop y. We mark those for deletion, which would give us
push x
push z DELETE
pop z DELETE
push y DELETE
pop y DELETE
pop y

We then try to check outside our optimization. We notice that pop y is on the bottom side, but

pop z is on the top. However, pop z is marked for deletion, so check above it. Push z is also marked for deletion, so skip over that; we finally find push x. We then check if push x pop y is an optimization pattern. It is, so apply the transformation:
movq x y
push z DELETE
pop z DELETE
push y DELETE
pop y DELETE
pop y DELETE.

Now we're done, and can delete everything marked for deletion. This is correct and compresses 6 wasteful instructions into a single, useful one!

Your compiler should include a "full optimizations" command-line option (see below). Your written documentation should present a detailed discussion of this option including how you determined the order in which your optimizations are performed and how many times you apply the optimizations. Finally, describe any hacks or solutions to tricky problems that you encountered.

For the full optimizations command line option, we had to decide in what order we would execute our optimizations. The optimizations we applied, and their order, were:
DCE→ CSE+Constant Folding→ DCE→ RegAlloc → Assembly Modification.

First, DCE goes over the base version of the code, removing any dead, unnecessary statements and declarations. CSE then runs, generating compiler generated temps for all the remaining things which, since DCE was run first, we know to be alive. DCE then runs again, cleaning up any unused compiler generated temporary variables. Next, we run register allocation on our representation of the code structure to assign variables to registers and/or spill them onto the stack. We then generate assembly code based on the program representation at this point, utilizing the register allocation results. After we generate the assembly code, we then directly optimize the assembly by replacing wasteful instructions or transforming push-pop relocation into simple moves. The assembly modification code is heavily optimized so that it can perform all possible optimizations, even ones that open up because of an optimization it will perform that has not yet occurred, in a single pass over the assembly instructions. After we optimize the assembly in this way, we print the final version of the assembly to the output file.

A tricky problem was that the result of register allocation was not directly integrated into our existing structure for assembly generation, so a hack was performed to manually transform the result into something from which we could generate assembly.

Overall, this has been one of the most challenging projects all of us have faced. The amount we learned from makes the amount of work we put in worthwhile.