http://6.035.scripts.mit.edu/fa17/handout-pdfs/07-optimizer-project.pdf
OUTLINE

- optimizations considered and how we decided which to include
- optimizations implemented: show beneficial, general, and correct; include before and after IR or assembly samples; empirical evidence that it sped things up
- hacks or solutions to tricky problems
- full optimizations option: describe what it does, how we decided the order to perform optimizations and how often to apply them

6.035 Optimizer Project
randomcompilers: arkadiyf, jamb, mayars
12 December 2017

1 Optimizations Considered

Our group considered implementing many of the optimizations discussed in class: copy propagation (CP), common subexpression elimination (CSE), dead code elimination (DCE), register allocation (RA), method inlining (IL), algebraic simplification (AL), strength reduction (SR), moving of loop-invariant code (LIC), and canonicalization of expression orders (CEO). We also considered assembly simplification (AS) by directly modifying lines to remove redundant operations.

We thought that all these optimizations would be useful, and tried to do as much as we could in the time available. CP, CSE, and DCE shared a common interface, so having written DCE for our dataflow project, it was easy and efficient for us to add the others. We noted that these optimizations couldn't do much for simplifying code across methods, so we added IL for non-recursive methods so that these optimizations could be more effective and so that we didn't need to push/pop for calls as often. We predicted that AS and CEO would help make CSE more effective by making more subexpressions the same, but didn't have time. We prioritized RA because looking at our code, many of the instructions seemed to be moving data between stack locations and registers, and so we wished to store more in registers. SR includes a number of ideas, but we mostly just considered reducing division and multiplication by powers of two to shift operations. Discussion of the design and benchmarking of these optimizations is included in the next section.

We started work on AS because we noticed that our assembly output included many lines like "mov -8(%rbp), %r10; mov %r10, %r11" that could easily be reduced to "mov -8(%rbp), %r11" (assuming %r10 and %r11 are not being used for longer-term value storage, which our program does assume), or even some eliminable lines like "mov %r10, %r10". After handling these two cases, the number of lines in our assembly shrank by 30-40%. However, after performing register allocation, registers *were* being used as storage for values, so AS was invalid. Due to

last-minute difficulties with RA and the derby, we did not have time to fix AS, so we basically removed it (though the removing trivial movs is still in place).

Examples of how the various optimizations benefitted us:

IL:
codegen 05-calls includes the following lines

```
int sum ( int a, int b, int c, int d, int e, int f, int g, int h ) {
  return ( a + b + c + d + e + f + g + h );
}
printf ( "sum of 1 - 8 is %d (36)\n", sum ( 1, 2, 3, 4, 5, 6, 7, 8 ) );
```

Method inlining initially just replaces the sum function with the addition, allowing us to avoid pushes/pops for the call. Combined with other optimizations such as copy propagation, it works even more magic: we can now avoid having variables declared for these numbers, which saves even more space.

CSE:
in optimizer tests such as saman_sl, there are heavily-repeated loops with lines like

```
image[i*303+j] = image[i*303+j+30];
```

CSE would allow the program to compute the common expression “i*303+j” just once per loop instead of twice, saving a multiplication and an addition for every single loop.

DCE and DVE:
saman_sl computes some unnecessary things. For example, it has the line

```
size = cols * rows;
```

but then never uses the computed value for size. DCE and DVE would remove the unnecessary computation and unused variable. In practice this optimization is less useful on large programs because the vast majority of the code is in loops which are generally live; and users tend not to write pointless code.

SR:
many programs have statements like:

```
x = y / 8;
```

Multiplications and divisions are hard, but with powers of two, we can rely on the easier shift operations (>> or <<). So the above statement would become x = y >> 3.

2 Optimizations Implemented -- Total Speedup 1.58

We examine the benchmark performance of our optimizations on optimizer tests to determine their usefulness. We noticed that there are many outliers (i.e. the code sometimes randomly takes a while longer to run), so for each optimization, we ran the test script 5 times and took the

average times for each test case. Then we computed the average speedups for each test and averaged those, getting the speedup values that we report here.

2.1 CP - Speedup 1.03

- Ran into problems with .equals because equality for two expressions at the CFG level was different than at the assembly level
- Kept a class CPDefinition that had a string (defined) and an expression (definition), and had a map to it that said whether or not this was a valid replacement
- Propagated a definition iff (1) it was the only definition of the string and (2) it was a valid definition, where arrays were not propagated.

2.2 CSE - Speedup 0.98

- Not included because nondeterministically failed on optimization tests and also seemed to slightly slow things down.
- Followed the same style as the other CFG optimizations and the algorithm given in class

2.3 DCE + DVR - Speedup 1.01

- Followed the dce algorithm given in class. Statements of the form a = method_call() were replaced by method_call()
- Had trouble deciding whether to remove globals.
- DVR = dead variable removal, prevents us from allocating space that we never use -- unused params, globals, or locals. This was particularly useful for cases such as f(a, b, c, d, …) { return a; } which previously had trouble having enough colors for register allocation.

2.4 RA - Speedup 1.42

Of our optimizations, the most impactful and easiest to justify was register allocation. This allowed us to store some values in registers rather than on the stack, lowering the number of memory accesses needed. The benefit of this is very clear in almost all code examples, where many to most of the memory accesses are removed. To implement this, we did liveness analysis on the CFG to figure out which pairs of variables were alive at the same time. Given this conflict graph, we colored the vertices such that no adjacent vertices had the same color, and then assigned registers based on color, moving things to the stack if there were not enough colors. Although doing this optimally is in NP, we used the greedy algorithm described in class, giving a suboptimal solution but finding it quickly. With this, the rest of the work only involved writing methods for accessing and moving data, which was largely similar to past work.

2.5 Inlining - Speedup 1.00

With inlining, we took nonrecursive methods and, wherever they were called, simply put the method code into that spot. The motivation for this was that method calls carry quite a bit of overhead in manipulating the stack and various pointers, as well as keeping track of caller and callee saved registers, and by skipping these we can ignore a lot of overhead. In implementing this, we implemented a method to copy a CFG while maintaining its contents such that they

could be inserted into another CFG. Because nonrecursive methods run exactly once, there is no difference between jumping to that method versus simply putting in all the commands in place of the method call.

2.6 Strength Reduction of Shifts -- Not fully implemented

This was one “optimization” which was not fully realized. Our progress here has consisted solely of taking all multiplications or divisions by powers of 2 and converting them to the equivalent shifts. Because this did not end up being a substantial change, it is on by default, even in the unoptimized code. So, this became a minor feature that code be extended into a more complete optimization, but was not implemented to this extent. As a result, we do not have any metrics on its impact on the code.

3 Problems Encountered and Ways Around Them

In working on the optimizations, most interacted poorly with arrays and globals. This was the cause of many bugs, and in general we found we had to be much careful dealing with these than with other parts of the optimizations. Additionally, register allocation was not only the most effective speedup, but also encountered another major difficulty. Not only were arrays and globals difficult, as their presence made it much more difficult to move information around, but following the standard method call convention was also very hard as well. Because we were storing values in registers, we occasionally needed to permute the values in the registers in order to properly call a method. Given more time, one better solution would have been to let this influence the choice of registers allocated to each variable, but this was far more complicated than we could work with.

Another major cause for added work was that register allocation needed to be toggleable. That is, we needed to maintain a way to run the code with no allocation as well as the more efficient code. To work around this, we defined the interface CFGLocationAssigner, which captured all the necessary information for manipulating data and keeping track of where it was stored. Getting it to this format took a bit of refactoring with the main body of assembly generating code, and was a bottleneck for our other work.

One change which was largely successful was a decision we made to include another layer between the CFG and the final output. We defined AssemblyLines, and stored a list of these before converting to just the code. Doing so opened up some opportunities to simplify code in some simple ways during this step. For instance, before register allocation, it could check for chains of movs and concatenate those if it found that we were doing extra work. We were initially hesitant about adding this level of abstraction, but looking back, we are very happy to have done it.

Finally, we also encountered an extremely nasty bug in compiling the unoptimized derby program which took a very long time to discover. Earlier, we had overridden .equals() for

IRExpressions, but the way this was done caused it to be possible for multiple expressions to get assigned the same temporary name, breaking the code. This was impossible to catch in smaller test cases, as it seems like the collision necessary did not happen, and even in the derby inserting trivial blocks of code hid the errors somewhat. Once we found this, we moved the necessary use of the equals implementation to another function and undid the changes to it. With this, we were able to continue comparing expressions while also not introducing devious collision bugs into the rest of the code.

4 Order of Optimizations

Our inlining of methods depends only on whether the method is recursive, which is not changed by the other optimizations, and it can allow other optimizations to do more since they can act on the inlined method in its context within the program. Therefore we run method inlining first.

Register allocation goes near the end, because it should only be run when all of the variables have been finalized (and ideally, some eliminated) by the other optimizations. Registers should only be assigned once. Assembly simplification is the very last because it acts directly on the assembly structures already generated.

The dataflow optimizations (CP, CSE, DCE, and DVE) run in between inlining and register allocation. They may be able to reduce the code further based on each other's work, so we loop until all of them make no further changes.