Phase 5 Writeup: Optimizations

Noah Pauls, **P**hilip Murzynowski, **R**obert Durfee

Design

So far, we have implemented the following optimizations:

- Global common subexpression elimination (`--opt=cse`)
- Global copy propagation (`--opt=cp`)
- Global constant folding (`--opt=cf`)
- Algebraic simplification (`--opt=as`)
- Dead code elimination (`--opt=dce`)
- Unused local elimination (`--opt=ule`)
- Unreachable code elimination (`--opt=uce`)
- Function inlining (`--opt=fi`)
- Register Allocation (`--opt=reg`)

You can enable all optimizations with `--opt=all`. They are applied in the following order (with steps 2-7 being repeated once):

1. Function inlining
2. Global copy propagation, global constant folding, algebraic simplification
3. Global common subexpression elimination
4. Global copy propagation, global constant folding, algebraic simplification
5. Dead code elimination
6. Unused local elimination
7. Unreachable code elimination
8. Register allocation

Note: CP is called twice given that our code generation naively creates a lot of copies that make global CSE much less useful. However, CSE also exposes a lot of copies which we want to eliminate, too. Hence the CP sandwich around CSE.

Global Common Subexpression Elimination

The global common subexpression elimination algorithm has two components, first, a global fixed-point working set approach computing the available expressions across basic blocks in a control flow graph, and second, a local value numbering based scheme for each basic block.

The global component is an optimistic forward data flow analysis, initializing all blocks except for the entry block to have all expressions available. In our low level intermediate representation binary, unary, and compare become our instructions of interest, as they compose expressions, and therefore can generate new expressions which must be set in our bitmap. However we chose not to potentially eliminate compare instructions as in our current architecture we use conditional jumps which are dependent on the flags register and therefore the comparison operation must be evaluated just before the jump disregarding possible manipulation of the flags register. To determine the availability of a given expression, a bitmap mapping a unique string describing an expression to a bit, 0 or 1, is used. Expressions are placed into the kill set whenever a variable used in an expression is redefined, or if the expression uses a global variable and there is a function call, which can potentially mutate the global variable. As a small side note, we may create a call graph to keep track of which global variables are used in function calls so that we do not have to be so conservative and kill all expressions involving global variables with each function call.

Once the global fixed point algorithm is complete, the local value numbering algorithm is run for each basic block is run, with the additional step of using the global bitmap if an expression is available from previous blocks.

Global Copy Propagation

The global propagation algorithm closely follows the one discussed in lecture. First, all reaching definitions are calculated by using a fixed-point working set approach. The working set initially holds all blocks. When any bitsets are changed, the successors are added to the working set.

Within each basic block, care must be taken with global variables. If we were being most conservative, we would not propagate any global variables as it is possible for a function call to happen at any point and modify a global variable. However, we found this to be too limiting as internal function calls are fairly rare and our naive code generation is pretty bad without any copy propagation. As a result, whenever an internal function call is encountered, all global definitions are reset to their identities so that globals are not propagated across function boundaries.

After all the reaching definitions are propagated, the transformation takes place. The transformation follows a fixed-point approach. However, whenever there is a change in any block, all blocks are traversed again. This accounts for the fact that a change in one block could potentially affect any of its descendants, not just its direct children.

Global Constant Folding

Within copy propagation, if at any point a binary or unary operator is visited that has all operands as constants, that instruction is replaced with the compile-time evaluation of that expression. If the evaluation yields a divide-by-zero exception, that exception replaces the expression instead.

Algebraic Simplification

Within copy propagation, the following algebraic simplifications are made:

- `a + 0, 0 + a => a`
- `a * 1, 1 * a => a`
- `a * 0, 0 * a => 0`
- `a / 1 => a`
- `a / 0 => DivideByZeroException`
- `0 / a => 0`
- `a % 1 => 0`

- `a % 0 => DivideByZeroException`
- `0 % a => 0`
- `0 - a => -a`
- `a - 0 => a`
- `a && true, true && a => a`
- `a && false, false && a => false`
- `a || true, true || a => true`
- `a || false, false || a => a`
- `a * 2^n, 2^n * a => a << n`

Note: The boolean simplifications are less helpful due to short circuiting as these operations do not appear directly in our LLIR. However, UCE should take care of this instead.

Dead Code Elimination

The dead code elimination algorithm follows exactly from the liveness analysis in lecture. We use a fixed-point, working set approach. The working set initially includes all basic blocks. Whenever a basic blocks entry set changes, all its predecessors are added to the working set. When no more changes occur, all basic blocks are traversed once again to remove any dead code using the liveness analysis. Special care is taken to make sure all global stores and array stores remain.

Unused Local Elimination

We sought to eliminate unused local variables primarily because previous optimizations generated a huge number of new local declarations. CSE, for instance, created a new local for every expression it attempted to eliminate, unnecessarily enlarging the stack with unused locals.

Within a single method, all uses of locals are collected. If a local present in the method’s list of locals is unused, that local is eliminated from the method and is not given space on the stack when the method is called. The exception to this is locals responsible for storing the “result” of void methods. While these are not technically used, they must be allocated for the function call to occur.

Unreachable Code Elimination

After CF, there are a large number of conditional branches where the comparison is between two constants. When simplifying the control flow graph, these branches are reduced to unconditional jumps and the unreachable blocks are automatically garbage-collected.

Function Inlining

While all other optimizations occur on the LLIR control flow graph, function inlining operates on the HLIR tree. It first passes over all methods estimating the number of instructions in each method which is considered the inline “cost”. Then, any function call with a cost below the threshold is labeled to be inlined.

When the LLIR control flow graph is being constructed, any function calls that are labeled to be inlined are copied directly into the control flow graph. The arguments are mapped to their corresponding temporaries and the local variable hoisting mechanism is easily reused. Whenever a return statement is encountered, the control flow is redirected to a single return target (like break and continue were handled previously).

Register Allocation

Register allocation was a priority in our phase 5 optimizations as direct computation on registers is far more performant than keeping all of the data allocated on the stack and loading to and from registers each time computation is needed.

We took a graph coloring approach, as after reviewing the linear scan algorithm we estimated that both approaches would be comparable amounts of working with limited code portability between the two approaches and we likely would not have time to try both, while in the literature we found that the linear scan algorithm could be around 20% slower than an aggressive coloring algorithm.

To perform the register allocation there were 5 main components: creating def-use chains from method control flow graphs, combining chains into webs assigning instructions to webs corresponding to the definitions and uses in an instruction, determining intersections between webs, and lastly assigning registers to specific webs (the coloring). To create def-use chains from the method control flow graphs we followed a similar approach to dead-code elimination, using a backwards analysis and identifying definition declarations with each use, and each use being identified in a method declaration by a triple formed of its basic block, instruction line, and index into list of uses in an instruction. To create webs we use a union find algorithm to group chains, which upon completion labels the sets of chains as a web, and then assigns instructions to the webs corresponding to their definitions and uses. After the webs have been created their interferences are determined by finding the interferences among consisting chains, and then they can be colored after an interference graph has been created, which is done using Chaitin Graph coloring algorithm and a simple heuristic is used to spill the web with with the greatest number of interferences.

Peephole Assembly Optimizations

Registers introduced the possibility of uses and defs for a single instruction utilizing the same register, leading to `movq` instructions where both operands were the same register (`movq %rax, %rax`). When generating move instructions, we eliminated any moves where both operands were the same location.

Our original assembly generator assumed that all operands originated in memory. Registers allowed us to reduce the number of instructions for various operations (such as `addq`) by checking if operands were present in registers at the beginning of the instruction. If they were, we could omit any instructions responsible for transferring a value from memory into a register before operating.

Because this optimization is part of what makes the results of register allocation efficient, it is included in the register allocation optimization by default.

Optimized Pushes/Pops

Register allocation required pushing/popping registers from the stack when making function calls. A naive approach was to push/pop all caller/callee saved registers for every call, but this turned out to be excessive; we improved this approach using the webs generated during register allocation, optimizing register pushes/pops at function calls using two methods.

First, only callee-saved registers that a function stores a new value in are pushed/popped. For each function, we find the webs associated with definitions in the function body. If those webs are colored with a callee-saved register, we know to store that register. Otherwise, the register isn’t stored.

Second, a caller-saved register is only stored if a web containing that register includes the relevant function call. During register allocation, we determine the active range of webs on a per-instruction basis, and we leveraged this information to determine what webs were active across function calls.

One optimization we did not perform was to omit the storing of caller-saved registers that the called function did not modify. However, this optimization proves complicated due to the presence of external calls whose modified caller-saved registers are not apparent, so we did not implement this optimization.

Because this optimization is part of what makes the results of register allocation efficient, it is included in the register allocation optimization by default.

Extras

Instead of just implementing one of CSE, CP, and DCE, we decided to implement all of them and add CF, AS, ULE, UCE, and FI as described above.

Void method return value elision

Previously, all function calls were given a storage location for their return value (in the stack or otherwise). This applied to void methods, despite their not having a return value. Internal calls returning void are no longer given storage for return values, slightly saving used memory.

Difficulties

The control flow graph simplification process is recursive. For very large programs, the large number of basic blocks results in stack overflow in our compiler due to the deep recursion (it is not an infinite loop). This severely restricts the size of programs we can compile. We will convert the process to be iterative to avoid this problem.

Other than that, we are not currently aware of any bugs that affect the correctness of the generated assembly. However, there are several optimizations we hope to improve.

We currently do not use CSE to the fullest extent possible. For example, consider the following:

```
 int a[100], i;
for (i = 0; i < 100; i++) {
  a[i] = 0;
}
```

In this code, the access `a[i]` can never be out of bounds because `i < 100` is available. Thus, the array out-of-bounds check `i < 100` is always true. We plan to make use of this in the future by replacing conditional jumps with unconditional ones and allowing UCE to take care of the rest.

We currently generate methods whether or not they are called. Now that we are inlining a lot of functions, we can clean up a lot of uncalled methods.

Both CSE and CP do not currently consider array accesses. However, it should be possible to extend both to perform CSE and CP on array accesses with constant indexing. This seems to be relatively common in some of the benchmarks.

We currently do not do any loop dataflow optimizations. We would like to implement some limited form of the following:

- Loop invariant code hoisting
- Loop unrolling
- Loop vectorization

Right now, CSE and CP are pretty conservative with globals. While we are not simply ignoring globals, instead resetting state upon internal function calls, this is still conservative as it assumes all globals are affected by every function call, which is unreasonable. When we do more analysis to remove uncalled methods, we plan to use this information to see which globals could be affected by a function call.

Currently, we only repeat our optimization passes once. However, we plan to update the optimizer to also be fixed point (i.e. run the sequence of optimizations until convergence).

FI does not allow inlining of any function with a nested internal function call. This is a very conservative way to avoid inlining a recursive function. This could be converted to a fixed-point algorithm instead.

DCE and UCE do not remove entire loops even if they do not yield any productive work. Once we implement some loop data flow optimizations, this will likely become more obvious.

There is still plenty of room for optimization. Our register allocator only uses spills (no splits) and the spill heuristic is simply the highest degree node that has not yet been colored.

Furthermore, we do not use precoloring as effectively as we could. Primarily, we use precoloring to place values in the appropriate function argument positions. For things like division operators, we simply ignore the `%rdx` register in allocation.

Also, our LLIR does not reflect our assembly generation that immediately moves all function arguments into the stack. As a result, all arguments, even with register allocation, are still accessed via the stack as it proved cumbersome to handle this in the web coloring.

When coloring webs, we do not use `%rax` or `%r10`. These registers are reserved for assembly generation. Ideally, the coloring should be aware of this and no registers should be reserved.

Contributions

We largely worked on different optimizations independently and debugged larger issues collaboratively (e.g. CSE and CP relating to globals). Noah worked on DCE and ULE, Philip worked on CSE, and Robert worked on CP (with CF and AS), UCE, and FI.

All three of us worked collectively on register allocation in group meetings. Noah further improved the excessive pushes/pops associated with caller-/callee-saved registers and function calls.