ECE 540 Project Report

Peter Goodman

April 12, 2012

1 Optimizations

I implemented six optimization passes, each of which appear as distinct files in the `lib/opt` directory. Each optimization is briefly described below.

1.1 Copy Propagation

Copy propagation (CP) was the simplest optimization to implement. This optimization can be disabled with the `ECE540_DISABLE_CP` flag.

CP tries to propagate register copies to the uses of registers in each instruction. A use of of register r_i can be replaced with a use of register r_j iff all definitions of r_i that reach the use of r_i are instructions of the form `cpy` r_i `=` r_k and $r_j = r_k$ for all k. Special consideration is made for temporary registers.

1.2 Constant Folding

The constant folding (CF) pass of this compiler performs more operations than a typical constant folder. This optimization can be disabled with the `ECE540_DISABLE_CF` flag.

CF folds constants in the usual way, and ignores floating point operations (except for `cvt` instructions). CF also performs local constant merging by combining multiple loads of the same constant within a given basic block. This does not limit the effectiveness of the CF pass because local constant copies are implicitly propagated through the CF pass' internal data structures. The CF pass maintains the property that constants are only ever loaded into temporary registers.

1.3 Dead Code Elimination

The dead code elimination (DCE) pass of this compiler operates in roughly the expected way, but instead of using post-dominators to track control dependencies, the DCE pass walks "backward" through the CFG. This is possible because the DCE pass maintains a work list of pairs of instructions and their containing basic blocks. This optimization can be disabled with the `ECE540_DISABLE_DCE` flag.

The DCE pass eliminates dead code, unreachable code, useless `jmp` instructions, and `nop` instructions. It considers function calls, memory stores, labels, and returns to be essential instructions.

1.4 Common Subexpression Elimination

The common subexpression elimination (CSE) pass of this compiler operates in the expected way. This optimization can be disabled with the `ECE540_DISABLE_CSE` flag.

The implementation of CSE was slightly complicated by the restrictions on temporary registers. If a temporary register computes a common subexpression then that register can only be used once, and only within the current basic block. When this case is detected, all uses and the definition of that temporary register are replaced with a new pseudo register, thus allowing the CSE pass to continue working.

Another interesting complication with CSE was the way I was maintaining the mapping of instructions to their available expressions, which operates on expression instances. Expression instances include a pointer to the instruction that computes the expression, the registers participating in the instruction, and the opcode of the instruction. When handling the case of a temporary register computing a common subexpression, it was necessary to "inject" new expression instances into the available expressions mapping to account for the replaced registers. This injection forces the the new instance to be equivalent to the old instance, thus maintaining consistency of the internal data structures used to track common subexpressions.

1.5 Loop-invariant Code Motion

The loop-invariant code motion (LICM) pass of this compiler operates approximately as described in class, except that it never changes the structure of the loop (e.g. zero iteration loop). This optimization can be disabled with the `ECE540_DISABLE_LICM` flag.

The LICM pass attempts to prove that the loop body will be entered using the symbolic interpreter (described below) with a breakpoint. This proof process operates in the following way.

First, we attempt to find a straight line sequence of basic blocks that leads into the loop header. In some cases, the best that can be done is finding the loop header itself. The motivation for finding the longest such straight line is so that the symbolic interpreter can build up more contextual information.

Second, we attempt to find a straight line sequence of blocks leading out of the loop header that reaches a conditional branch. If the branch keeps control flow in the loop or is a multi-way branch, then we assume that the loop cannot be proven to run. If the branch exits the loop, then the next instruction inside the loop is set as the breakpoint of the loop.

Finally, if the symbolic interpreter is able to reach the breakpoint then we say that the loop will provably run at least once. We only care about the first reached loop exit because code that is hoisted must dominate all loop exits.

Testing showed that most loops in the test cases were proven to be entered by the above process. In the event that a loop cannot be proven to run, a stronger condition is required: only instructions that dominate the exit block of the CFG are hoisted.

The motivation for not changing the loop's structure was partially due to bad data structure choices that might have lead to inconsistencies/inefficiencies in the LICM pass when operating on nested loops.

Loops are processed from the inside-out. This works by ordering loops according to the number of basic blocks in their bodies. While disjoint loops will be processed in any order, this guarantees that the partial order of loop nesting is followed.

Because each loop instance maintains a set of all blocks in the loop body, a change the structure of a nested loop would have required changes to all enclosing loop instances. In hindsight, a better design would have been to continually re-compute the set of basic blocks in the loop body.

1.6 Symbolic Interpretation

The symbolic interpretater (EVAL) pass of this compiler behaves in a similar way to constant folding and local value numbering. This optimization can be disabled with the `ECE540_DISABLE_EVAL` flag.

This optimization pass was specifically designed to target the `testprog.c` test case. The SimpleSUIF system does not (easily) enable interprocedural optimizations and so function inlining and compile-time evaluation of pure functions was not possible. However, there are a number of pure functions, some of which compute a value that is a simple/small expression parameterized over some of the function's arguments. Typically, what was seen was that there was a loop where only the last iteration of the loop was relevant to the output of the function.

EVAL operates by assigning a unique symbolic value to every defined/used register in a procedure. The symbolic value initially assigned to each register identifies that register. EVAL then attempts to evaluate each instruction. If a symbolic value participates in an expression (e.g. `r1` or `r2` in `add r3 = r1, r2`) then a new a symbolic expression is generated. If all registers (e.g. `r1` and `r2`) are compile-time values, then `r3` is given a concrete value that is the result of folding the computation. Constant loads and register copies operate in the expected way.

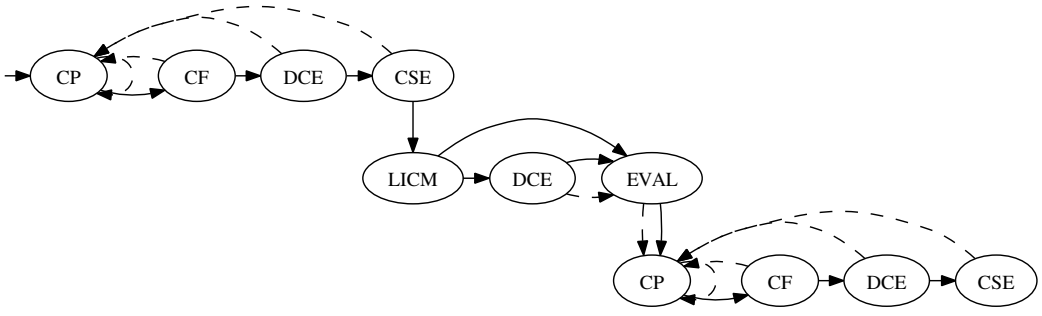
If EVAL encounters a conditional branch and cannot determine the next instruction then it gives up. If, during the interpretation of the procedure, a very large expression is computed (where large is the measured by the depth of the expression DAG) then the large expression is replaced with a dummy "big value" symbolic expression. This is to prevent the execution of some functions from being unrolled into sequences of hundreds or thousands of instructions.

If EVAL reaches a `ret` operation then it will try to emit a new instruction sequence for the returned registers. Instructions are emitted by performing a postorder traversal of the symbolic expression DAG assigned to the returned register.

More information on the EVAL phase (as well as a simulator of the EVAL pass) can be found in my blog post on the subject: `http://www.ioreader.com/2012/04/07/symbolic-interpretation`.

2 Optimization Pipeline

The following pipeline of optimization passes is run for every procedure:



Transitions within the pipeline are conditional on the source optimization having changed something. A dashed transition from A to B implies that pass A effected some form of change in the control-flow graph. A solid transition from A to B implies that pass A had no effect.

Individual compiler passes appear multiple times in the pipeline. Each instance of a given pass within the pipeline is considered unique. The ordering of optimization passes is significant. Most passes

tend to generate certain amounts of garbage, either in the form of `nop` instructions or by "orphaning" instructions (e.g. leaving a redundant or useless instructions in place). The purpose of DCE is to clean up this garbage.

Optimization passes are managed by the `optimizer` class. The `optimizer` class handles conditional cascading, dependency injection, and data structure tainting for each optimization pass. Optimization passes are registered with the `optimizer` class. Each registration represents a unqiue instance of each optimization pass. Cascading relationships are defined on these instances.

Optimization passes are represented by functions. The types of the arguments to an optimization function represent that optimization pass' dependencies. The `optimizer` class creates custom dependency resolution and injection code for each unique function type signature.

The first argument to every optimization pass is required to be a reference to the `optimizer` class. This is so that individual optimization passes can notify the pipeline when certain classes of changes have been made. The tracked changes include: inclusion of `nop` instructions, changing a register use, changing a register definition, and changing a basic block (to represent control flow changes).

Different changes cause different data structures to be tainted (potentially put into an inconsistent state). As well, different changes guide the cascading behaviour of the pipeline.

The dependency injection functions resolves issues of inconsistent data structures by re-computing only tainted data structures that are needed as arguments to the next compiler optimization pass.