Matthew Moore (mlm1), Guy Goldstein (guyg), Arthur O'Dwyer (ajo) 12/13/2004

# Optimization Write-Up

## 1 Overview

Starting this lab we faced two tasks: adding modules to our compiler which optimized the code it produced, and improving its efficiency so that it ran in a reasonable amount of time while doing so. We knew that on large tests, such as the quine (which originally produced about 37 000 lines of assembly on our compiler), liveness analysis was clearly the bottleneck. For the other aspect of the project, we knew we would have to complete at least the optimizations that the reference compiler was going to have, in order to compete. The optimizations we decided to include were

- Move Coalescing: to get rid of the excessive number of moves produced during translation and code generation.
- Dead Code Elimination: to get rid of instructions that would slow down the compilation process and the resulting code unnecessarily.
- Common Subexpression Elimination: to avoid recomputing expensive arithmetic and memory computations at runtime.
- Constant Propagation: to enable more dead code elimination and constant folding.
- Copy Propagation: to enable code hoisting, more dead code elimination and more CSE.
- Constant Folding: to avoid computation of expressions at runtime which could easily be computed at compile time.
- Unreachable Code Elimination: to avoid a lot of branching and reduce the size of the produced code considerably.
- Function Inlining: to avoid the high cost associated with a function call, and to enable all of the above optimizations that the separation of the function bodies would otherwise disable. (For example, we can fold constants across function-call boundaries with inlining.)

In addition to these optimizations, we played around with code hoisting in two forms: one for very busy expressions, and the other for loop invariant hoisting. The hoisting based on the very busy expressions dataflow analysis didn't have its intended effect, and so it was disabled for the submitted version. In some cases it was even slowing down the generated code! Loop invariant hoisting proved more difficult to implement than we had expected, and partly as a result our implementation of it ended up causing a slowdown in the generated code as well.

## 2 Move Coalescing

This optimization was implemented to rid our produced assembly of the superfluous move instructions that were inserted during both translation and code generation. We implemented a pretty standard move-coalescing algorithm; the main difference from the book's algorithm is that we don't do coalescing during the coloring phase. Based on our representation for the liveness graph, the

Briggs heuristic proved much faster than the George heuristic, and they produced comparable results, so rather than slow our compiler down unreasonably for little gain, we decided to stick with using just the Briggs heuristic for coalescing.

## 3 Dataflow Functor

To implement several of the remaining optimizations, we required a module which would solve "fixed point" equations given the appropriate lattice operations. To allow us to avoid rewriting a lot of code, we wrote a `Dataflow` functor, which takes in the various lattice operations and returns a module which computes the fixed point for the given statements.

## 4 Dead Code Elimination

This optimization was implemented to get rid of code that performs no meaningful computation (i.e., whose result is never used). We implemented this in two phases: one prior to code generation and one after liveness analysis. The intent of the first round is to clean up all of the garbage left over from other optimizations, such as definitions which are no longer used. This sped up the liveness analysis somewhat because it removed a considerable number of instructions from consideration. The second round is just an augmentation of liveness analysis. It uses the data computed during liveness to make one more sweep of the instructions and remove those that were dead at their creation.

The second round is slightly more aggressive. It is based on associating a reference count with each "definition," such that every "use" it reaches increments that count. Each definition which ends up with a reference count of 0 can be safely removed. However, we can decrement the reference count of each definition referenced by the removed definition. This allows some cascading, and in practice seems to remove a good number of definitions.

## 5 Common Subexpression Elimination (CSE)

This optimization was implemented to enable the elimination of recalculations. To implement this optimization, we first determine the "available expressions" in the program. We implemented a dataflow algorithm (using the `Dataflow` functor) for determining the available expressions at the various statements; once these are calculated, we scan the statements to see which available expressions are actually used. For those expressions we find to be useful, we perform the transformation given in the book: The expression is moved into a new temp everywhere it is generated, but not available, and we use that temp everywhere the expression is both available and generated.

## 6 Constant Propagation

This optimization was implemented to enable the elimination of variables whose sole purpose is to carry around constant expressions. This optimization also required the use of the `Dataflow` functor, because it uses the results of the reaching-definitions analysis. Once the reaching definitions have been computed, we scan through the instructions, and everywhere that a definition is used (where the definition is being assigned a constant), we replace that use with the constant and perform constant folding, so that we can keep propagating if the current definition becomes constant as a result. As a proof of concept, we wrote a program which took more than 20 seconds for the code produced by the reference compiler and fewer than 6 on ours, using only constant propagation and folding.

## 7 Copy Propagation

This optimization was originally intended to enable code hoisting. However, it provided enough of a speedup of its own that we kept it around even after we decided not to include code hoisting. Once we had constant propagation done, it was relatively trivial to implement copy propagation because the dataflow portion (reaching definitions) had already been implemented, and the code to process the results was fairly similar. The difference is that instead of replacing temps with constants, we replace them with temps (and there is no need to fold the constants afterward).

## 8 Constant Folding

An attempt was made to implement a slightly more powerful version of constant folding than the one described in class. The new method, based on rotations of the IR tree, was described in Arthur's homework 6, problem 4. The method employs a long list of tree rotations which essentially codify the arithmetic rules of commutativity, associativity, and distributivity of various operators; for example, the IR tree `BINOP(BINOP(`$x$`,PLUS,`$y$`),PLUS,`$z$`)` would be rotated by the constant folder to become `BINOP(`$x$`,PLUS,BINOP(`$y$`,PLUS,`$z$`))` whenever $x$ is a constant. Thus we accumulate constant terms in the left-hand leaf nodes near the root, where a few simple rules can fold them together.

This method is very fast, having running time about $O(n)$, but unfortunately it proved ineffective. By the time expressions make it to the IR with which the constant folder deals, they have been broken up into tiny pieces with a lot of intermediate assignments to temps; for example, the expression $x+y+z$ given above might really look like this in our IR: `ESEQ(MOVE(`$t_k$`,BINOP(`$x$`,PLUS,` $y$`)),BINOP(`$t_k$`,PLUS,`$z$`))`. The constant folder can't reconstruct the original, "foldable" expression at this point, so it loses a lot of its potency.

The code for performing these rotations remains enabled in our compiler, since the time it takes is negligible, but the more complicated rotations have very little, if any, effect on the generated code.

## 9 Unreachable Code Elimination

This optimization didn't require the use of any fancy flow analysis. It involves producing the reachability graph of the instruction list. We start at the program entry point, and recursively mark all of the instructions which are reachable from that point. Then we filter out all of the unmarked program points. This method proved incredibly efficient and effective.

## 10 Function Inlining

To allow for easy function inlining, we changed the way that functions are translated. When a function's body is translated, it relies on several "parameter" temps, and it eventually returns (potentially with some value). We modified the structure of the final translation so that instead of handling multiple `return` statements by generating multiple `RET` instructions, it sets some temp $t_r$ to the value at each return point and then jumps to some end label.

With this setup, we take the translated function body and wrap it in an SML function. This function takes in a list of expressions to assign to each of the parameters and also the name of the temp $t_r$ to which we will assigned the function's return value. Now, to create the "actual" function body, we call this function on a list of the memory locations as offsets from `%ebp`, e.g., `[MEM3(%ebp, 0,0,8),MEM3(%ebp,0,0,12)]` for a two-parameter function. We also pass in a new temp $t_r$. The function will produce a statement $s$. We put this in sequence with a return: `SEQ(`$s$`,RETURN(SOME` $t_r$`))`. This resulting sequence does what the function "should" do, which is read the parameters from memory and yield a value in $t_r$.

This new framework allows the easy inlining of functions—after surmounting one small technical difficulty: making all of the internal labels and temps "fresh." Once this is done, for some fresh temp $t_k$, we can change expressions of the form `CALL(f,params)` into expressions of the form `ESEQ(ff params` $t_k$`, TEMP` $t_k$`)`, where `ff` is the SML inlining function for the translated L3 function named `f`.

If done universally, inlining can lead to code explosion; therefore, we need *good* heuristics for which functions to inline, and in which locations. We tried several heuristics, all inlining only a single level of function calls. They included:

- Inline all function calls
- Only inline functions shorter than a certain length
- Only inline function calls within loops
- A combination of the above using the metric $D \cdot C < \mathit{thresh} \cdot (L+1)$, where $D$ is the approximate length of the function body, $C$ is the number of calls to that function in the current function body, *thresh* is a tunable quantity, and $L$ is the number of loops we are currently inside.

We found that for some of the test cases, inlining significantly improved our results. In the test case `cmccabe-jtran`, which ran in roughly 19 seconds on the `unix` servers, universal inlining got it down to 14 seconds pretty consistently. Similarly, for the test `bgilbert-sgowal`, which ran in 1.7 seconds with no inlining, inlining improved the generated code to 1.3 seconds. However, the test `3.l3`, which ran in 3.84 seconds on the `unix` servers, was slowed to 4.6 seconds, and in the tests `4.l3` and `5.l3`, the code explosion slowed the compilation process down enormously, to the point where it exceeded the time limit.

None of the heuristics we tried could achieve the best of both worlds, but it seemed like the final heuristic came the closest to approximating it. Unfortunately, it caused our own test's compile time on the `unix` servers to go from roughly 50 seconds to 90 seconds, which we knew would be far longer than the 3 minute limit on the system used to compute power rankings. However, using this heuristic, we were able to get the total time of the power rankings down from about 68 seconds to 61 seconds, so the performance improvement is clear. To get around this, we decide to impose a hard growth limit on function inlining. We have a fixed bound on the size of the generated instruction list for any program, above which no function inlining will occur. With this limit in place, all the other tests took roughly the same amount of time as they had without it, but the compile time of *our* test case dropped back to about 50 seconds. In a more modern compiler with analysis techniques such as Basic Block Tuning (BBT), it would be fairly easy to tell which function calls occur the most and inline those, but, alas, we are not that advanced.

## 11 Code Hoisting

For the very-busy-expression–based code hoisting, the intention was to hoist computations that occur along every branch from a conditional jump point. It was good that we attempted this optimization because it led us to write copy propagation, but in retrospect, if a computation is performed along every path from a given point, and you hoist it above the branch and put copy instructions where the expression used to be computed, then along every path, we are now both computing that instruction and copying it, so we are actually making a trace through the program longer. Also, we are making shorter the basic blocks to which the branch leads, which isn't good on modern architectures.

Loop invariant hoisting turned out little better. When code was lifted, it increased the liveness range of variables, which wound up forcing more spills of temps during coloring; and some of those temps wound up being used in the loops out of which the code had been lifted in the first place. Thus instead of doing simple computations inside the loop we were doing memory reads and writes, which turned out to be quite a bit slower. We probably could have developed a reasonable heuristic for lifting any code that was sufficiently slow, even if it caused spilling, but doing general lifting the way we implemented it turned out to hurt the generated code, so it was not included in the final version.

## 12 Optimizing Translation and Code Generation

Several minor changes in our translation into IR and from IR to assembly produced code that was much shorter and easier to optimize. The most significant changes were in the representation of `NULL`, pointer arithmetic, and indexing of arrays.

In the P3 version of our compiler we represented `NULL` as any pointer with an address and size of 0, but then had to do extra `NULL` checks in our code. Also, pointers were uniquely bound to their 12-byte fat pointer space, which meant that moving pointers required moving all 12 bytes of the source pointer's memory into the destination pointer's 12 bytes of memory. We eased this restriction and made it so that new 12-byte spaces are generated for each pointer arithmetic operation, but multiple pointers can share the same fat pointer memory. Also, we originally implemented indexing of an array as pointer arithmetic followed by a dereference; this caused each index to require allocation and copying to 12 new bytes of heap space. We fixed this by making our dereference code take a third expression: the offset from the pointer that should be dereferenced. In general this made our IR code substantially shorter and thus sped up our optimizations. It also produced assembly code with fewer jumps and memory operations, speeding up the generated code.

## 13 Retrospective

Some things that we would change if we had it all to do again:

- For legacy reasons, we inserted bounds checking during the translation phase of our compiler, so CSE was unable to eliminate repeated bounds checks. For example, the bounds check on `a[i]` would not be eliminated in the following code.

  ```
  x = a[i]->f1;
  y = a[i]->f2;
  z = a[i]->f3;
  ```

  In retrospect, we would probably change this so that CSE would be performed after translation, but before the bounds checks were inserted.

- The AST is represented by a tree data structure of the form

  ```
  datatype exp' = ExpConst of const
                | ExpBinop of exp * binop * exp
                ...
  and const = const' Mark.marked
  and exp = exp' Mark.marked
  ```

  where the `Mark.marked` wrapper contains tagging information such as the position in the input file where this expression occurs. This is a good representation for many purposes, but it requires many "helper" functions in the code dealing with ASTs, and—most unfortunately for the constant folder—prevents SML's native pattern matching on AST values, because the opaque `Mark.marked` type gets in the way. A more felicitous representation might have been, as in Arthur's implementation,

  ```
  datatype exp = ExpConst of const * tagging_info
               | ExpBinop of exp * binop * exp * tagging_info
               ...
  ```

  This representation has its own irritating qualities, of course. Perhaps the best solution would be the creation or discovery of a programming language which contained explicit language-level support for this kind of "tagged tree" data structure.

- After the optimizations, we can end up with a lot of blocks of the form

  ```
  .L42:
  jmp .L33
  ```

  We could quite easily use a union-find to sweep across the produced code and, in this case, union the labels `L42` and `L33`, and remove this pair of instructions entirely, replacing jumps to `L42` with jumps to `L33`.

- We left a lot of room for optimizing our compiler. One example would be to implement a `BitArray` structure which handles sparse bit arrays well as the current `BitArray` structure handles dense bit arrays well.

  Another space for improvement is the way we are doing liveness analysis, which would be improved dramatically by making it worklist-based rather than modification-based. We currently re-run liveness as long as something changes; it would be good to keep track of what changed to we don't have to do a lot of needless recomputation.