le02 Optimization Project Proposal

We considered the following optimizations when planning the project; each optimization is listed with an owner for the optimization, whether the optimization will be done before or after the first project checkpoint, and the goal of the transformation.

Additionally, we assigned bug-fixing and infrastructure changes as appropriate; although these are not optimizations specifically, we feel they should be included in the proposal to explicitly account for the distribution of work.

Bugfixes/infrastructure

- **Fix over-optimism in CSE** [dkoh/lizf] [done by checkpoint]
  Our common sub-expression elimination is too aggressive and is causing corruption in the blur.dcf test program. We will isolate this problem and fix it.
- **Fix integer width and floating point operations** [lizf] [done by checkpoint]
  Our divide instruction is failing in philbin.dcf due to attempting to divide 0x000000000[...]0 in %rdx, 0xFFFF[...]1234 in %rax by 1; the dividend overflows the appropriate 64-bit register and triggers a FPE. We will solve this problem by handling all integers as 32-bit numbers instead of 64-bit numbers and by performing sign extension where appropriate. This bug also causes checks such as '(2147483647 + 1) == -2147483648' to fail.
- **Automatically run and validate decaf test programs on test inputs and compare to golden outputs** [lizf] [done by checkpoint]
  Our presubmit needs to be changed to only allow the submission of code that does not break programs that run correctly unoptimized.

Already implemented optimizations

- **Common subexpression elimination** [lizf/dkoh]
  Stores results of intermediate computation to temporary variables and utilizes temporary variables instead of recomputing if subsequent subexpressions match the intermediate results and the computation is still valid.
- **Copy propagation** [lizf]
  Allows for more efficient dead code elimination by converting references to copied variables to the original source variable if the original variable is unmodified at time of read of the copy.
- **Dead code elimination** [kshaunak]
  Eliminates instructions that write to variables that are not subsequently read.

Considered/proposed optimizations

- **Expression canonicalization** [mfrend] [done by checkpoint]
  We propose to standardize the ordering of operations in all complex expressions at the IR level in order to ensure that CSE is as effective as possible in identifying identical subexpressions.
  Example:
  x = a + c + z + d => x = a + c + d + z
  y = c + a => y = a + c
  This allows the subexpression a + c to be detected and only computed once by CSE.
- **Constant folding** [kshaunak] [done by checkpoint]
  After running loop unrolling and loop-invariant code motion, we expect to create a number of expressions which are arithmetic functions of constants. By computing these expressions during compile time, we can eliminate operations which would otherwise be performed at runtime.
- **Eliminate unnecessary array bounds checking** [kshaunak]
  Right now, array bounds checking code is generated during ASM generation. By the checkpoint, we will move this code into the CFG so that it can be optimized. Afterwards, we will try a number of strategies for eliminating as many checks as possible, including the following:
  a. Elimination of bounds check when the array index is a loop index variable with a value that is guaranteed to be within the bounds
  b. Elimination of checks when a variable is used to index into the array twice without being redefined
  c. Some amount of sign prediction to eliminate lower bounds checks
- **Peephole optimizations**:
  a. **Eliminate duplicate MOV instructions** [lizf] [done by checkpoint]
     We currently perform duplicate MOVs when we are storing the result of a variable assignment to a temporary in addition to the target variable. Specifically, we currently output for spilled variables:
     ```
     tempreg := a op b
     var.memoryloc := tempreg
     tempreg := var.memoryloc

     tmp.memoryloc := tempreg
     ```
     We intend to reduce this to:
     ```
     tempreg := a op b
     var.memoryloc := tempreg
     tmp.memoryloc := tempreg
     ```
     Or for ordinary register operations:
     ```
     var.reg := a op b

     tmp.reg := var.reg
     ```
     This eliminates an extra memory operation, in addition to any duplicated array bounds checks.

   b. **Single-instruction strength reduction** [lizf] [done by checkpoint]
      Strength reduction replaces expensive single operations with less expensive equivalents to those operations. For example, 2^n * x could be replaced with x << n, which avoids the expense of a MUL operation. x / 2^n could be replaced with x >> n. x % 2^n can be replaced with x & (2^n-1). We can also implement LEA multiplication (http://www.godevtool.com/GoasmHelp/progtips.htm#lea).

- **Register allocation** [dkoh] [done by checkpoint]
  We plan to do register allocation using a heuristic graph coloring algorithm. In order to do this, we will use a three step approach.
   a. Calculate webs for every variable in the code
   b. Generate an interference graph from those webs
   c. Using a graph coloring algorithm, we will attempt to color the interference graph using available registers.

  By the checkpoint, we will have a register allocator which spills registers when it cannot fit all live values into existing registers. After the checkpoint, we will add static evaluation of spill costs to attempt to choose the spilled register intelligently (prioritizing register uses in loops over out of loops). If there is time, we will also attempt to split webs using this statically calculated spill cost heuristic.
- **Loop invariant code motion** [lizf]
  Assignments whose values are constants across different iterations of a loop and that are always executed within the control flow of the loop can be identified and pulled outside the loop to be executed exactly once rather than being recomputed at each loop iteration. In order to do this, we need to analyze what variables change as a result of control flow transfers (e.g. invalidating expressions containing globals if a loop may contain a function call, and invalidating expressions that depend upon the loop index or other index-dependent expressions). We also must ensure that any expressions we pull outside the loop are unconditionally executed for all loop iterations. Finally, we must ensure that the loop invariant code's output variable is only written once in the loop.
  Examples:
```
for i = 0, 10 {
  x = a + b;
  y = a + b;
  if (i < 5) {
    z = c + d;
  }
  k = a + i;
  y = c + d; }
```
  In these examples, the statement involving x is the only statement that can be hoisted.
  y cannot be hoisted since its value changes within the loop (however, the values a+b and c+d could be precomputed and stored in temps prior to the loop, and swapped into y as appropriate).
  z cannot be hoisted because it only is executed on some code paths through the loop.
  Any other expression containing i, y, or z cannot be hoisted.
- **Elimination of empty blocks** [kshaunak] [done by checkpoint]
  Right now, some of our optimizations delete every statement in a basic block. During code generation, we then write a jump into these empty blocks, immediately followed

by a jump out of them. By traversing the control flow graph and removing empty basic blocks, we can eliminate these jumps.

- **Removal of unnecessary JMPs and basic block ordering** [lizf]
  Our code currently performs a direct jump at the end of every basic block to the following basic block (even when the code does not logically branch). We will order the basic blocks such that most of the time, the code can simply continue from one basic block to the next, and only jump/branch when necessary.
  Examples:
```
.block0:
  movq %rax %rcx
  jmp .block1
.block1:
  movq %rdx %r11
  [...]
```
  becomes
```
.block0:
  movq %rax %rcx
.block1:
  movq %rdx %r11
  [...]
```
- **Parallelization** [dkoh]
  We plan to do outer-loop parallelization on our code. We will do this in a multi-step process. The first step will be the loop analysis, which will already be done by the loop induction variable code. The second step will be the calculation of distance vectors for each loop iteration, with the help of the given parallel analysis library. Finally, we will determine how many threads to parallelize the loop iteration between (this will take into account the maximum number of cores available, which is 4 in the derby environment, and also the amount of parallelism the loop contains, as indicated by the calculated distance vector). Using a thread pool, we will generate code to perform the loop calculation in parallel.
  We are not planning to perform parallelization on inner loops due to the increased overhead and finer granularity of work done in inner loops. Though there are code transformations which could reduce this work, the complexity of performing these transformations leads us to believe it would not be worthwhile given our work timeframe.
- **Instruction reordering** [mfrend] [done by checkpoint]
  Our current implementation does not order the assembly instructions in a way which encourages pipelining. We plan to reorder the instructions so that instruction stalls are minimized. We will create a dependence DAG and implement the list scheduling algorithm, as described in class, to manage data dependencies, control dependencies, and resource constraints.
- **Loop unrolling** [mfrend]
  We plan to implement loop unrolling to further take advantage of the processors pipe-lining capabilities. As described in class, we will unroll a few iterations of the loop, which will be processed in different sets of registers, and use this to find a steady-state. We will the generate the necessary pre- and post-ambles.
- **Loop induction variable** [mfrend]
  We plan to identify induction variables and use them to perform strength reductions and to eliminate dependant induction variables.

- **SMD/SSE** [lizf] [not implementing]
  We believe that SMD/SSE would potentially show benefits for some of the image manipulations as they perform operations upon successive elements of an array; however, the optimizations are difficult to perform general pattern-matching upon, and are only easily applicable to transforms that access data in the same row but not data in vastly different rows. Thus, we are not planning on performing SMD/SSE initially unless we have extra time at the end of the project.

Full optimizations

We intend to run the optimizations in the following order:
(Scanning/Parsing)
(IR generation)

1. Expression canonicalization

(Semantic checks)
(Control Flow Graph generation)

1. Copy Propagation
2. Common Subexpression Elimination
3. Dead Code Elimination
4. Array bounds check elimination
5. Loop induction variable detection/elimination
   Parallelization detection
6. Loop invariant code motion
7. Constant folding
8. Elimination of empty basic blocks
9. Register Allocation

(Code generation)

10. Parallel loop code generation
11. Unnecessary JMP removal and proper ordering of basic blocks (done during codegen)
12. Instruction peephole optimizations (e.g. strength reduction, elimination of duplicate MOVs)
13. Instruction reordering