SSA based global code motion of identical computations

ADITYA KUMAR, Samsung Austin R&D Center
SEBASTIAN POP, Samsung Austin R&D Center

We present a global code motion compiler optimization which schedules identical computations across the program so as to save code size. Not only this code motion saves code size, it exposes redundancies in some cases, it exposes more instruction level parallelism in the basic block when the computations are hoisted, and it enables other passes like loop invariant motion to remove more redundancies. The cost model to drive the code motion is based on live range analysis on SSA representation such that the (virtual) register pressure does not increase.

We have implemented the pass in LLVM. It is based on Global Value Numbering infrastructure available in LLVM. The experimental results show an average of 2.5% savings in code size in llvm testsuite, although the code size also increases in many cases because it enables more inlining. We have also seen imrpovements of approximately 5% in a couple of SPEC 2006 benchmarks viz. gcc and mcf, moreover, register spills reduced for almost all the SPEC 2006 bencmarks when compiled for X86_64. This is an optimistic algorithm in the sense that it considers all identical computations in a function as potential candidates. We make an extra effort to hoist candidates by partitioning the potential candidates in a way to enable partial hoisting in case common hoisting points for all the candidates cannot be found. We also formalize how register pressure will reduce as a result of code-motion and why sorting the list of potential candidates w.r.t. their depth first numbers helps hoist more candidates with less compile time overhead.

CCS Concepts: •**Software and its engineering** → **Compilers; Source code generation;** •**General and reference** → **Performance;**

Additional Key Words and Phrases: Optimizng Compilers, GCC, LLVM, Code Generation, Global Scheduling

ACM Reference format:
Aditya Kumar and Sebastian Pop. 2017. SSA based global code motion of identical computations. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 17 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Compiler techniques to remove redundant computations are composed of an analysis phase that detects identical computations in the program and a transformation phase that reduces the number of run-time computations. Classical scalar optimizations like CSE (Aho et al. 1986) work very well on single basic blocks but when it comes to detect redundancies across basic blocks these techniques fall short: more complex passes like GCSE and PRE have been designed to handle these cases based on dataflow analysis (Morel and Renvoise 1979). At first these techniques were described in the classical data-flow analysis framework, and later the use of the SSA representation lowered the cost in terms of compilation time (Briggs and Cooper 1994; Chow et al. 1997; Kennedy et al. 1999) and brought these techniques in the main stream: nowadays SSA based PRE is available in every industrial compiler.

This paper describes code-motion of identical computations, a technique that uses the information computed for PRE to detect identical computations but has a transformation phase whose goal differs from PRE: it removes identical computations from different branches of execution. These identical

with paper note.
2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

computations in different branches of execution are not redundant computations at run-time and the number of run-time computations is not reduced. It is not a redundancy elimination pass, and thus it has different cost function and heuristics than PRE or CSE. It is more similar to global code scheduling (Aho et al. 1986; Click 1995) in the sense that it will only move computations. We will now consider an example that explains how code hoisting can also improve performance.

1.1 Illustrative Example

Code hoisting can reduce the critical path length of execution in out of order machines. As more instructios are available at the hoisting point, the hardware has more instructions to reorder. Following example illustrates how hoisting can improve performance by exposing more ILP.

```
float fun(float d, float min, float max, float a) {
  float tmin, tmax, inv;

  inv = 1.0f / d;
  if (inv >= 0) {
    tmin = (min - a) * inv;
    tmax = (max - a) * inv;
  } else {
    tmin = (max - a) * inv;
    tmax = (min - a) * inv;
  }
  return tmax + tmin;
}
```

In this program the computations of tmax and tmin are identical to the computations of tmin and tmax of sibling branch respectively. Both tmax and tmin depends on inv which depends on a division operation which is generally more expensive than the addition, subtraction and multiplication operations. The total latency of computation across each branch is: $C_{div} + 2(C_{sub} + C_{mul})$ Or, for out of order processors with two add units and two multiply units: $C_{div} + C_{sub} + C_{mul}$

Now if the computation of tmax and tmin are hoisted outside the conditionals, the C code version would look like this:

```
float fun(float d, float min, float max, float a) {
  float tmin, tmax, tmin1, tmax1, inv;

  tmin1 = (min - a);
  tmax1 = (max - a);

  inv = 1.0f / d;
  tmin1 = tmin1 * inv;
  tmax1 = tmax1 * inv;

  if (inv >= 0) {
    tmin = tmin1;
    tmax = tmax1;
  } else {
    tmin = tmax1;
```

```
    tmax = tmin1;
  }

  return tmax + tmin;
}
```

In this code the two subtractions and the division operations can be executed in parallel because there are no dependencies among them. So the total number of cycles will be $max(C_{div}, C_{sub})+C_{mul} = C_{div}+C_{mul}$; since C_{div} is usually much greater than C_{sub} (ARM 2014; Intel 2000).

In fact there are several advantages of code-motion of this kind:

- It reduces the code size of the program;
- It improves function inlining heuristics: functions become cheaper to inline when their code size decreases because inliner heuristics in LLVM depends on instruction count.
- It expose more instruction level parallelism to the later compiler passes. By hoisting identical computations to be executed earlier, instruction schedulers can move heavy computations earlier in order to avoid pipeline bubbles;
- It reduces branch misprediction penalty on out-of-order processors with speculative execution of branches: by hoisting/sinking expressions out of branches, it can effectively reduce the amount of code to be speculatively executed and hence reduce the critical path;
- It reduces interference/register pressure when appropriate cost-model is applied. We have used a cost model in our implementation which hoists/sinks only when register pressure would potentially reduce.
- For SIMD architectures, which execute all branches, it will reduce the total number of instructions to be executed.
- It improves passes that do not work well with branches:
  - to improve loop vectorization by reducing a loop with control flow to a loop with a single BB, should all the instructions in a conditional get hoisted or sinked;
  - to enable more loop invariant code motion (LICM): as LICM does not reason about instructions in the context of loops with conditional branches, code-hoisting is needed to move instructions out of conditional expressions and expose them to LICM.

There have been a lot of work both in industry and academia to hoist and sink code out of branches, and in general global scheduling (Click 1995). Some relate code-hoisting to code-size optimization (Rosen et al. 1988) and many (Barany and Krall 2013; Shobaki et al. 2013) use global scheduling to improve performance. Most of the recent work on global scheduling are done using ILP which results in prohibitively high compile time. To the best of our knowledge we have not found any reference which explored code-motion of identical computation in as much detail as we have done. A part of our implementation (aggressive code hoisting) is already merged in LLVM trunk, however, a general implementation of code-motion of identical expressions is still missing from GCC and LLVM trunk. The main contributions of this paper are:

- a new optimistic algorithm to move computations out of branches,
- a cost model to reduce interference and reduce spills,
- a technique to maximize hoisting in an optimistic approach by partitioning the list of potential candidates sorted by their DFS visit number,

- experimental evaluation of our implementation in LLVM which combines SSA based liveness analysis, and ranking of expressions to move very busy expressions in order to reduce code-size (and improve performance in some cases).

2 RELATED WORK

There are a lot of bug reports in GCC and LLVM bugzillas (GCC 2016a; LLVM 2016), showing the interest in having a more powerful code hoist transform. The current LLVM implementation of code hoisting in SimplifyCFG.cpp is very limited to hoisting from identical basic blocks: the instructions of two sibling basic blocks are read in the same time, and all the instructions of the blocks are hoisted to the common parent block as long as the compiler is able to prove that the instructions are equivalent. This implementation does not allow for an easy extension: first in terms of compilation time overhead the implementation is quadratic in number of instructions to bisimulate and second, the equivalence of instructions is computed by comparing the operands which is neither general nor scalabale.

Dhamdhere (Dhamdhere 1988), Muchnick (Muchnick 1997) mention code hoisting in a data flow framework. A list of Very Busy Expressions (VBE) computed which are hoisted in a basic block where the expression is anticipable (all the operands are available). This algorithm would hoist as far as possible without regarding the impact on register pressure and as such a cost model will be required. Also the description of VBE is based on the classic dataflow model and an adaptation to a sparse SSA representation is required.

Rosen (Rosen et al. 1988) also introduced moving computations from successors but using value numbering. Their algorithm iterates on computations of same rank and move the code with identical computations from the sibling branch to a common dominator if they are very busy (Muchnick 1997). We also incorporate ranking of expressions in our implementation.

GCC recently got code-hoisting (GCC 2016b) which is implemented as part of GVN-PRE: it uses the set of ANTIC_IN and AVAIL_OUT value expressions computed for PRE. ANTIC_IN[B] contains very busy expressions at basic block B i.e., values computed on all paths from B to exit and AVAIL_OUT[B] contains values which are already available. The algorithm hoists top down to a predecessor. It uses ANTIC_IN[B] to know what expressions will be computed on every path from B to exit, and can be computed in B. It uses AVAIL_OUT[B] to subtract out those values already being computed. The cost function is: for each hoist candidate, if all successors of B are dominated by B, then we know insertion into B will eliminate all the remaining computations. It then checks to see if at least one successor of B has the value available. This avoids hoisting it way up the chain to ANTIC. It also checks to ensure that B has multiple successors, since hoisting in a straight line is pointless. The algorithm continues top down the dominator tree, iterating with PRE until no more changes. One advantage of GCC implementation is that it works in sync with the GVN-PRE such that when new hoisting opportunities are created by GVN-PRE, code-hoisting will hoist them.

Click (Click 1995) describe aggressive global code motion to first schedule all the instructions as early as possible. This results in very long live ranges which is mitigated by again scheduling all the instructions as late as possible. They report sppeedup of as high as 23%.

Barany (Barany and Krall 2013) presented a global scheduler with ILP formulation with a goal to minimize register pressure. The results they got were not very promising. It may be because they only used the scheduler for smaller functions (< 1000 instructions) also they compiled the benchmarks for ARM-Cortex which is more resilient to register pressure because it has more registers compared to say X86.

Shobaki (Shobaki et al. 2013) also recently presented a combinatorial global scheduler with reasonable performance improvements. It is possible that both Barany and Shobaki's implmentation will have similar results when compiled for same target. Also, both suffer from the same problem, although Shobaki not so much, of large compile times which is not feasible for industrial compilers like gcc and LLVM.

We got reduction in register spills on some SPEC2006 benchmarks, reduction in code-size and some performance improvements with very low compile time overhead.

3 CODE MOTION

The algorithm for code motion uses several common representations of the program that we shortly describe below:

- Control Flow Graph (CFG) and the Dominance (DOM) and Post-Dominance (PDOM) relations (Aho et al. 1986);
- DJ-Graph (Sreedhar 1995) is a data structure that augments the dominator tree with join-edges to keep track of data-flow in a program. We use DJ-Graph to compute liveness of variables as illustrated in (Das et al. 2012).
- Static Single Assignment (SSA) (Cytron et al. 1989);
- Global Value Numbering (GVN) (Click 1995; Rosen et al. 1988): to identify similar computations compilers use GVN. Each expression is given a unique number and the expressions that the compiler can prove to be identical are given the same number;
- Memory SSA (Novillo 2007): it is a factored use-def chain of memory operations that the compiler is able to prove are dependent.

The code-motion pass can be broadly divided into the following steps that we will describe in the rest of this section:

- find candidates suitable for code-motion
- compute a point in the program where it is both legal and profitable to move the code,
- transform the code and update data structures to continue iterative code motion.

3.1 Liveness using DJ Graph

DJ-Graph (Sreedhar 1995) is a data structure that augments the dominator tree with join-edges to keep track of data-flow in a program. We use DJ-Graph and Merge Sets to compute liveness of variables as illustrated in (Das et al. 2012). It is very efficient for computing liveness and does not require any bitvectors to be maintained for each basic block. The underlying simplicity of liveness computation is due to SSA form, where the the values only flow (from def to use) either through dominator edges or the join edges (where we insert a PHI). The DJ-graph contains both these edges which allows for computation of merge sets for each basic block i.e., a set of all basic blocks where the values can flow from a particular basic block. We have implemented merge-set computation from DJ-graph as illustrated in (Das and Ramakrishna 2005). A simplified version of merge set that we implemented is presented here:

```
// Return true if the merge set of source node of
// a visited J-edge (incoming edge of lnode) is not
// the subset of the merge set of lnode.
bool still_inconsistent(Node lnode, JEdges JE, Visited V, MergeSet M)
   for (all incoming edges to lnode) do
     Let e = Incoming edge
     if (e is in JE && e in V) then
       Let snode = Source Node of e
       if (M(snode)!(Subset) M(lnode)) then
         return true
```

```
      end if
    end if
  end for
return false
// Compute merge set top-down in breadth first order.
bool constructMergeSet_1(BFSList B, JEdges JE, DomLevel DL)
repeat = False
// List of visited edges
Visited V
for n in B do
  for e in (all incoming edges to n) do
    if (e is in JE && e not in V) then
      V(e) = true
      Let snode = Source Node of e
      Let tnode = Target Node of e
      Let tmp = snode
      Let lnode = NULL
      while (DL(tmp)  DL(tnode)) do
        Merge(tmp) = Merge(tmp) U Merge(tnode) U {tnode}
        lnode = tmp
        tmp = dom-parent(tmp)
      end while
      repeat = still_inconsistent(lnode, JE, Merge)
    end if
  end for
end while
return repeat

// Construct Merge set of each node in control-flow-graph G of a function.
void  constructMergeSet(CFG G) {
  B = Breadth First Order of G
  JE = JEdges of G
  DL = List of Path length (from root) of each node in G.

  do // Call until a fixed point is reached.
    Repeat = constructMergeSet_1(B, JE, DL);
  while (Repeat);
}
```

We compute merge sets of the control flow graph which remains same through the code-motion transformation. For code-motion we only want to know if a use operand is a kill (to compute changes in register pressure). For that we only need to know whether the use is also required later in the execution path. A variable is live out of a basic block B if it is used in the merge set of B. We compute the live-out relation on-demand when profitability of hoistable/sinkable candidates is to be evaluated. A simplified version of isLiveOutUsingMergeSet is presented here:

```
bool isLiveOutUsingMergeSet()
```

```
// Compute if variable a is liveout from basic block n
Input: Node N, Variable a

if a is defined in N then
  if a is used outside any basic block other than N then
    return true;
  else return false;
endif

Ms(n) = null;
// Mergeset of N is the union of merge sets of its successors
for w in successors(n) do
  Ms(n) = Ms(n) U Mr(w);
endfor

// Iterate over all the uses of a and see if any intersect with the
// merge set of N.
for t in users(a) do
  b = basic_block(t)
  while (b != null) and (b != def_bb(a)) do
    if b  Ms(n) then
      return true;
    endif
    b = dom-parent(b);
  endwhile
endfor
return false;
```

The original algorithm presented in (Das et al. 2012) has mismatched types in terms of uses and nodes because each node can have many instructions and hence many uses. Also, while iterating on the dominator of each user of a variable we may reach to the beginning of a function, in that case the inner while loop needs to terminate. These two cases were missing from the algorithm and we came across them during implementation.

3.2 Finding candidates to move

The first step is to find a set of instructions that perform identical computations: this is performed by a linear scan of all instructions of the program and classifying all instructions by their value given by GVN. We could compute available and anticipable sets as computed by GCC's code-hoisting but that would be a lot of data structures to maintain at each basic block level. For GCC it makes sense because their code-hoisting is integrated with GVN-PRE which already has those data structures available.

The current implementation of GVN in LLVM has some limitations when it comes to loads and stores so we compute the GVN of loads and stores separately. Our solution to value number loads is to hash the address from where the value is to be loaded. For stores, we value number the address as well as the value to be stored at that address. Another limitation of the current GVN implementation in LLVM is that the instructions dependent on the loads will not get numbered correctly, and so after hoisting all candidates we need to rerun the GVN analysis in order to discover new candidates now available after having hoisted load instructions. This limitation should be addressed in a new implementation of the GVN based on MemorySSA, that would better account for equivalent loads and their dependent instructions.

The process of computing GVN can be on-demand (as we come across an instruction) or, precomputed (computing GVN of all the instructions beforehand). Which process to choose is determined by the scope of code-hoisting we want to perform. In a pessimistic approach, we want to hoist a limited set of instructions from the sibling branches as we iterate the DFS tree bottom-up, it is sufficient to compute GVN values on-demand. Whereas, in the optimistic approach, as described in Section 4.1, we want to move as many instructions as possible, and it would require GVN values to be precomputed.

Once the instructions have been classified into equivalence classes, we compute for each group of equivalent instructions, a point in the program that is both legal and profitable for the instructions to be moved to.

3.3 Legality check

Since the equality of candidates is purely based on the value numbers, we also need to establish if hoisting them to a common dominator or sinking them to a common post-dominator would be legal. Once a common dominator (post-dominator) is found, we check whether all the use-operands of the set of instructions are available at that position. It is possible reinstantiate (remateralize) the use-operands in some cases when the operands are not available and make it legal to move the instruction.

Subsequently, it is checked that the side-effects of the computations (if any) do not intersect with any side-effects between the instructions to be hoisted/sunk and their hoisting/sinking point. It is also necessary to check if there are indirect branch targets e.g., landing pad, case statements, goto labels etc., along the path because it becomes difficult to prove safety checks in those cases. In our current implementation we discard candidates on those paths.

Code-motion basically consists of two parts i.e., hoisting and sinking. The unique part about this algorithm is that it only moves identical computations. An immediate guarantee of this approach is gain in code size (the final executable may be of larger size because of more inlining). The algorithm prefers hoisting to sinking. If the dependency of a hoistable candidate is in the same basic block as the candidate, then the dependency must also be hoistable otherwise hoisting will be illegal or would require a complicated code generation to make it legal. The current algorithm discards cases if the dependency is neither hoistable nor rematerializable.

3.3.1 Legality of hoisting scalars. Scalars are the easiest to hoist because we do not have to analyze them for aliasing memory references. As long as all the operands are available (or can be made available by rematerialization), the scalar computations can be hoisted.

3.3.2 Legality of hoisting loads. The availability of operand to the load (an address) is checked at the hoisting point. If that is not available we try to rematerialize the addresss if possible. Along the path, from current position of the load instruction backwards on the control flow to the hoisting point, we check whether there are writes to memory that may alias with the load, in which case we discard the candidate. We use the MemorySSA infrastructure of LLVM to iterate on the use-def chains available for memory references.

3.3.3 Legality of hoisting stores. For stores, we check the dependency requirements similar to the hoisting of loads using the MemorySSA of LLVM. We check that the operands of the store instruction are available at the hoisting point, that there are no aliasing loads or store along the path from the current position to the hoisting point.

3.3.4 Legality of hoisting calls. Call instructions can be divided into three categories: those calls equivalent to purely scalar computations, calls reading from memory, and most of the time, without further information, calls have to be classified as writing to memory, that is the most restrictive form. Each category of call instructions is handled as described for scalar, load, and store instructions.

3.3.5 Legality of sinking expressions. One might wonder that if a set of expressions compute the same value then why not just hoist them. The problem is, sometimes, hoisting is not upward-safe **??** e.g., if the expressions are in a landing pad etc. Sometimes sinking may reduce the register pressure e.g., when the use operands are not kills.

For sinking, higher ranked expressions would be sunk first. And it would be illegal to sink higher ranked identical expressions if they are not anticipable in the common post-dominator. For example:

```
B0: i0 = load B
B1: i1 = load A
    c1 = i1 + 10
    d1 = i0 + 20
    goto B3
B2: i2 = load A
    c2 = i2 + 10
    d2 = i0 + 20
    goto B3

B3: phi(c1, c2)
    phi(d1, d2)
```

In this example (c1, c2) or (d1, d2) could be sunk first. Since (c1, c2) depend on i1 and i2 respectively which are also in their original basic blocks, c1 and c2 are not anticipable in B3. So without knowing the sinkability of 'i1' and 'i2' it would be illegal to sink (c1, c2) to B3. On the other hand (d1, d2) can safely be sunk because their operands are readily available at the sink point i.e., B3. It should also be noted that, just because the expressions are identical and operands are available, it still requires a unique post-dominating PHI to use the exact same values to be legally sinkable.

A general global scheduling algorithm also requires checks for undefinedness when introducing a new computation along a path. Since only very busy expressions are moved in the current implementation, there is no need to check for undefinedness resulting due to movement of instruction. This simplifies the implementation.

3.3.6 Partitioning the list of hoistable candidates to maximize hoisting. In the approach, described in Section 4.1, it is possible that a common hoisting point of all the instructions is either too far away, or not legally possible. In these cases, it is still possible to 'partially' hoist a subset of instructions by splitting the set of candidates and finding a closer hoisting point for each subset.

In order to hoist a subset of identical instructions, we partition the list of all candidates in a way to maximize the total number of hoistings. By sorting the list of all the candidates in the increasing order of their depth first search discovery time stamp (Cormen et al. 2001) (DFSIn numbers), we make sure that candidates closer in the list have their common dominator nearby. In Figure-1, if B3, B4, B5, and B6 have identical computations and for some reason they cannot be hoisted together at B0, then we partition the set of hoistable candidates in their DFSIn order. In this case the DFSIn ordering would be B3, B4, B5, B6 which will allow the instructions in B3, B4 to be hoisted at B1 and those in B5, B6 to be hoisted at B2.

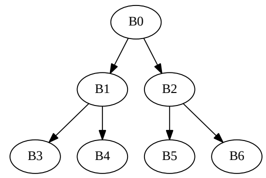

Fig. 1. CFG to illustrate partitioning

In our current implementation we keep as many candidates in one set as possible (greedy approach). We split the list at a point where the legality checks fail to hoist subset of candidates which are legal to hoist and then start finding new hoisting point for the remaining ones.

3.4 Profitability check (Cost models)

After the legality checks have passed, we check if a code-motion is profitable. That takes into account the impact code-motion would have on various parameters that might affect runtime performance e.g., impact on live-range, gain in the code size. Since this is mostly a code-size optimization pass, the goal is to not regress in performance across popular benchmarks at the same time reduce code size as much as possible. Following are the cost models which are implemented:

3.4.1 Reduce register pressure. Hoisting upwards will decrease the live-range of its use, if it is a last use (a kill), but increase the live-range of its definition. Conversely, sinking will decrease the live-range of the defined register but increase the live-range for killed operands. If the live-range after code-motion is less than before it will be moved.

Essentially, as long as there is one killed operand , code hoisting will either decrease or preserve the register pressure. Similarly, code-sinking will either decrease or preserve the register pressure as long as there is one operand killed at most.

Following example explains how code motion of identical computations can reduce the register pressure. Consider the following example where the labels prefixed with 'P' represent the position of instruction in a basic block (names prefixed with 'B').

```
    b = m
    c = n
B0: if c is true then goto B3 else goto B4

B1: a0 = b<kill> + c<kill>

B2: a1 = b<kill> + c<kill>

After hoisting a0 and a1 are removed and a copy
of a0 as a01 is placed in B0 just before the branch.
```

In this case, since 'b' and 'c' are killed in 'a0' and 'a1', hoisting them will reduce the register pressure in B3 and B4 because two registers will be freed.

Ideally, it should be okay to hoist all the instructions and a later a live-range-splitting (Cooper and Simpson 1998) pass should make the right decision of rematerializing the instruction should it be beneficial to do so. But the current live-range splitting pass of LLVM is not making the optimial decision and we have found regressions while hoisting aggressively.

Moreover, LLVM has a 'getelementptr' instruction which computes the address where a load or a store would happen. It is a scalar computation and gets hoisted frequently even if the loads/stores would not get hoisted. In order to reduce register pressure while hoisting loads, we have restricted hoisting of address computations away from their corresponding loads and stores when the loads and stores cannot be moved. This restriction is only to mitigate the limitations of LLVM's register allocator and may be lifted in the future, when the register allocation rematerialization pass has been improved to catch the regressions.

3.4.2 Hoisting expression across call. Even hoisting scalars across calls is tricky because it can increase the number of spills. During the frame lowering of calls, the argument registers, in general, the caller saved registers are saved because they might be modified by the callee and after the call they are restored. So before the call, the register pressure is high because the number of available registers are reduced by the number of caller saved registers. In that situation if a computation that increases register pressure is not profitable to hoist.

Hoisting loads/stores across calls also require precise analysis of all the memory addresses accessed by the call. The current implementation being an intraprocedural pass, cannot hoist aggressively across calls. In the presence of pure calls, loads can be hoisted but stores can't. Also, if the call throws exceptions, or if it may not return, nothing can be hoisted.

3.4.3 Profitability of hoisting scalars. Since scalars are the majority of instructions which are hoisted, we pay special attention in case of hoisting scalars too far, as that may increase register pressure and result in spills. For example hoisting a scalar past a call, as described in Section 3.4.2. In our current implementation we hoist scalars past a call only when optimizing for code-side (-Os). Ideally, a later stage of live-range splitting pass should split the live-ranges for optimal performance, however, that is not the case with LLVM as we have found regressions when scalars are hoisted too far, as in Section 3.4. Another way to mitigate this problem is be to reinstantiate (rematerialize) the computation after a call (may be as a different optimization pass).

3.4.4 Profitability of hoisting loads. A load instruction introduces a register where the value loaded will be kept, the register pressure increases by one (unless the operand to load becomes dead at the load). On the other hand, loading a value early will reduce the stall during execution should the value is not in the cache. We generally prefer to hoist load except when the hoisting point is too far (this distance is computed by looking at the experimental results of representative benchmarks, see Section 5).

3.4.5 Profitability of hoisting stores. Since stores do not increase the live-range of any registers, and in some cases it ends the liveness of registers, we hoist all the stores.

3.4.6 Profitability of hoisting calls. Currently we hoist all the calls that are suitable candidates for hoisting.

Since stores and calls are hoisted the least **??** the performance does not change much whether they are hoisted or not.

3.5 Code generation

Once all the legality and profitability checks are satisfied for a set of identical instructions, they are suitable candidates for hoisting. A copy of the computation is inserted at the hoisting point along with any instructions which needed to be rematerialized. Thereafter, all the computations made redundant by the new copy are removed, and the SSA form is restored by updating the intermediate representation

(IR) to reflect the changes. At the same time MemorySSA is also updated to get up-to-date information about memory references.

After one iteration of algorithm runs through the entire function, it creates more oppportunities for *higher ranked* computations (Rosen et al. 1988). Currently, this is a limitation of the GVN analysis pass, and so we rerun the code-hoisting algorithm until there are no more instructions left to be hoisted. Obviously, this is not the most optimal approach and can be improved by ranking the computations (Rosen et al. 1988), or by improving the GVN analysis to correctly number loads and dependent instructions.

Finally after the transformation is done, we verify a set of post-conditions to establish that program invariants are maintained: e.g., consistency of use-defs, and SSA semantics.

4 SSA BASED GLOBAL CODE MOTION OF IDENTICAL COMPUTATIONS

The amount of hoisting depends on whether we collect GVN of instructions before finding candidates (optimistic) or, on-demand (pessimistic). It also depends on the generality of the GVN algorithm. We have implemented a optimistic global code motion of identical computation which uses the liveness analysis as illustrated in Das (Das et al. 2012) and ranking expressions explained in Rosen (Rosen et al. 1988).

4.1 Optimistic code motion algorithm

We collect the GVN of all the instructions in the function and iterate on the list of instructions having identical GVNs. The algorithm prefers hoisting to sinking. So first we find the common dominator dominating all such identical computations and perform legality checks, as described in Section 3.3. Often times it is not possible to hoist all the instructions to one common dominator, due to legality constraints, e.g., intersecting side-effects, or profitability constraints, e.g., increasing register pressure. In those cases, this algorithmn would partition (Section 3.3.6) the list of identical instructions into subsets which can be partially hoisted to their respective common dominators. However, it should be noted that we only hoist very busy expressions to avoid checking for undefined behaviors resulting because of introducing extra computation in an execution path.

computing barriers:

```
computing downward-safety of hoistable instructions at hoist-point
Worklist = list of all basic blocks of hoistable instructions
for each basic block B in the dominator tree starting at hoist-point:
  if Worklist is empty
    return false // Path exists! not downward safe
  if B is in Worklist:
    remove B from worklist
    remove subtree with root at B // Available at B => downward safe from B
  if B is a leaf basic block:
    return false  // Path exists! not downward safe
  if B dominates hoist-point:
    return false  // Back edge

Analyses available: Dominator Tree, DFS Numbering, Memory SSA,

Compute GVN of each expression in the function
Compute DJ Graph of function and compute mergeset based on that
```

```
For each value number VN with 2 or more instructions:
  sort the instructions according to DFSIn numbers
  if first two I1, I2 are hoistable:
    if they are downward-safe at hoist-point:
      if they are profitable to be hoisted at hoist-point:
        proceed to check hoistability of subsequent instruction with same VN
      else
        proceed to check if I2 and I3 (if available) is hoistable

For each hoistable VN:
  move the first one I1 to the hoist-point
  update the use of all other candidates to refer to I1
  remove all others.
  if I1 has memory references (a load, store etc.)
    update MemorySSA of I1 and others to point to the MemorySSA reference of I1
    remove MemorySSA reference of all others which were deleted
  update statistics

If any of candidates were hoisted then repeat hoisting
Once no more candidates are hoistable proceed to sinking
Analyses available: Dominator Tree, DFS Numbering, Memory SSA, Mergeset
Compute GVN of each expression in the function

For each value number VN with 2 or more instructions:
  if there are two instructions with same immediate successor as post-dominator
    if both are only used in the same PHI-node of the successor
      if both have dependencies that are available or can be made available at the sink point
         mark VN as sinkable

For each pair I1, I2 of sinkable instructions:
  Move the first one I1 to the sink point which is just after all the PHI-nodes
  in the post-dominator
  update the use of PHI with I1
  Remove I2 and PHI
  if I1 is a memory reference:
    update MemorySSA of I1 and others to point to the MemorySSA reference of I1
    remove MemorySSA reference of all others which were deleted
  update statistics
```

Code hoisting opens new opportunities for other hoistable candidates which were of higher rank (depended on candidates which got hoisted). Ideally we could iterate on lower ranking expressions first and then proceed to higher ranking expressions in the same iteration but LLVM's GVN infrastucture does not compute equivalence classes in a effective way. We found it simpler to just recompute the value numbers and start finding hoistable candidates again.

4.2 Time complexity of algorithm

The complexity of code hoisting is linear in number of instructions that could be hoisted in the program, matching the complexity of PRE on SSA form. The analysis phase is based on the Global Value Numbering (GVN), the same analysis used for PRE, followed by the computation of a partition of identical expressions to be hoisted in a same location to guarantee safety properties and program performance, and followed by a simple code generation that adds the identified instruction in the hoisting point and removes all the now redundant expressions.

5 EXPERIMENTAL EVALUATION

We ran LLVM test-suite (trunk:d87471f8) with the patch (trunk:86940146). All the experiments were conducted on x86_64 Ubuntu-Linux machine and at -Ofast optimization level. The results for code-hoisting are listed in Table 1. The table lists the number of scalars, loads, stores and calls hoisted as well as removed. For each category, the number of instructions removed is greater or equal to the number of instructions hoisted because each hoisting is performed only when at least one identical computation is found.

Loads are hoisted the most followed by scalars, stores and calls in decreasing order. This was the common trend in all our experiments. One reason why loads are hoisted the most is the early execution of this pass (before mem2reg) in the LLVM pass pipeline. Passes like mem2reg, instcombine might actually remove those loads so this order may change should this pass be scheduled later.

Metric	Number
Scalars hoisted	6791
Scalars removed	9696
Loads hoisted	14802
Loads removed	20719
Stores hoisted	15
Stores removed	15
Calls hoisted	8
Calls removed	8
Total Instructions hoisted	21616
Total Instructions removed	30438

Table 1. Code hoisting metrics on LLVM test-suite

Other static metrics are listed in Table 2. Here we can see that except for rematerializing defs for splitting, which has an overhead of 2%, all other parameters have less than 1% overhead. This is to explain why the performance does not go down with our implementation (and cost-model) of code hoisting pass.

While benchmarking LLVM test-suite we see both increase as well as decrease in the codesizes of the final binaries. Since the pass runs early, it affects many optimizations which rely on the number of instructions, lenth of the use-def chain, and other metrics. For instance, the inliner is impacted by a decrease in the number of instructions in the caller and callee, as its heuristics estimate the size of functions to be inlined. Various code-size metrics are shown in Table 4. All but one benchmark varied between -5.32% and 5.43%. In one benchmark FreeBench/distray/distray.test, the codesize increased by 35.38%. In this benchmark 3 more functions got inlined (15 as compared to 12) and because of that

Metric	Before	After
Call sites deleted, not inlined	1988	1988
Functions deleted (all callers found)	38250	38255
Functions inlined	154986	154985
Allocas merged together	212	212
Caller-callers analyzed	193042	193092
Call sites analyzed	414336	414381
Rematerialized defs for spilling	18321	18326
Rematerialized defs for splitting	5719	5842
Spill slots allocated	42912	42970
Spilled live ranges	61330	61362
Spills inserted	50724	50784

Table 2. Static metrics before and after code-hoisting on LLVM test-suite

Code-size metric (.text)	Number
Total benchmarks	497
Total gained in size	39
Total decrease in size	58
Median decrease in size	2.9%
Median increase in size	2.4%

Table 3. Code size metrics on LLVM test-suite

10 more vector instructions got generated (81 vs. 71), 3 calls got hoisted/sunk as (compared to 0), one loop got unswitched (compared to 0), 6 high latency machine instructions got hoisted out of loop, 59 (compared to 30) machine instructions got hoisted out of loop, 70 (compared to 39) machine instructions were sunk.

The code shown in Section 1.1 is a reduced example that appears in a hot loop of a proprietary benchmark. When the expressions are hoisted from the conditional clauses, the overall performance of that benchmark improves by 15% on an out-of-order processor due to increased instruction level parallelism, and better scheduling of the instructions, accommodating for the long latency of the division operation.

6 CONCLUSION AND FUTURE WORK

We have presented the GVN based code hoisting algorithm. The primary goal is to reduce the code size but it benefits performance in some cases as well. To preserve performance and not hoist too much we have implemented several cost models described in Section 3.4. Since those cost models depend on a set of thresholds, it requires tuning, as such, we used representative benchmarks to tune them.

In general it is a good idea to start with lower ranked expression first such that maximum hoisting can happen in one iteration, however, current implementation does not rank the expressions and iteratively finds a fixed point when no more candidates are available. Even this implementation converges quickly and no significant compile time regression have been observed because of code hoisting pass. This is not the most optimal approach and results in multiple data structures to be recomputed. This can

Number of spills	base	code-motion	%loss
400.perlbench	2542	2481	0.97
401.bzip2	718	707	0.98
403.gcc	5778	5710	0.98
429.mcf	14	17	1.21
433.milc	616	624	1.01
444.namd	3185	3223	1.01
445.gobmk	2166	2170	1.00
447.dealII	10038	9151	0.91
450.soplex	1104	1114	1.00
453.povray	5199	5176	0.99
456.hmmer	1195	1195	1
458.sjeng	168	169	1.00
462.libquantum	183	183	1
464.h264ref	3382	3423	1.01
470.lbm	30	30	1
471.omnetpp	521	522	1.00
473.astar	176	196	1.11
482.sphinx3	603	594	0.98
483.xalancbmk	5101	4941	0.96
Grand Total	42719	41626	0.97

Table 4. Number of spills on SPEC2006

be improved by ranking the computations (Rosen et al. 1988). Also GVN-hoist runs very early in the pass pipeline, it will be good to evaluate the codesize/performance impact when it is run in sync with GVN-PRE just like GCC does.

With the implementation of code-hoisting in LLVM, the passes which rely on the code-size/instruction-count to make optimization decisions needs to be revisited. The first candidate would be the inliner. We have seen different inlining decisions in Table 4, before and after code-hoisting was enabled. Since inliner has several magic numbers tuned for the previous pass layout, it would need some improvement.

ACKNOWLEDGMENTS

We would like to thank Daniel Berlin for his code reviews and for his feedback on earlier versions of this paper and Brian Grayson for motivating examples that started this work.

REFERENCES

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. *Compilers, Principles, Techniques*. Addison wesley.

ARM. 2014. ARM Cortex-A57 Software Optimization Guide. (2014). http://infocenter.arm.com/help/topic/com.arm.doc.uan0015b/Cortex_A57_Software_Optimization_Guide_external.pdf

Gergö Barany and Andreas Krall. 2013. Optimal and heuristic global code motion for minimal spilling. In *International Conference on Compiler Construction*. Springer, 21–40.

Preston Briggs and Keith D Cooper. 1994. Effective partial redundancy elimination. In *ACM SIGPLAN Notices*, Vol. 29. ACM, 159–170.

Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. 1997. A new algorithm for partial redundancy elimination based on SSA form. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 273–286.

Cliff Click. 1995. Global code motion/global value numbering. In *ACM SIGPLAN Notices*, Vol. 30. ACM, 246–257.

Keith D Cooper and L Taylor Simpson. 1998. Live range splitting in a graph coloring register allocator. In *International Conference on Compiler Construction*. Springer, 174–187.

Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. 2001. *Introduction to algorithms*. Vol. 6. MIT press Cambridge.

Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 25–35.

Dibyendu Das, B Dupont De Dinechin, and Ramakrishna Upadrasta. 2012. Efficient liveness computation using merge sets and dj-graphs. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 27.

Dibyendu Das and U Ramakrishna. 2005. A practical and fast iterative algorithm for φ-function computation using DJ graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005), 426–440.

Dhananjay M Dhamdhere. 1988. A fast algorithm for code movement optimisation. *ACM SIGPLAN Notices* 23, 10 (1988), 172–180.

GCC. 2016a. GCC bugs related to code hoisting. Bug IDs: PR5738 , PR11820, PR11832, PR21485, PR23286, PR29144, PR32590, PR33315, PR35303, PR38204, PR43159, PR52256. (2016). https://gcc.gnu.org/bugzilla

GCC. 2016b. GCC code hoisting implementation. (2016). https://gcc.gnu.org/viewcvs/gcc?view=revision&revision=238242

Intel. 2000. Intel IA-64 Architecture Software Developer's Manual. (2000). http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. 1999. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999), 627–676.

LLVM. 2016. LLVM bugs related to code hoisting. Bug IDs: 12754, 20242, 22005. (2016). https://llvm.org/bugs

Etienne Morel and Claude Renvoise. 1979. Global optimization by suppression of partial redundancies. *Commun. ACM* 22, 2 (1979), 96–103.

Steven S. Muchnick. 1997. *Advanced compiler design implementation*. Morgan Kaufmann.

Diego Novillo. 2007. Memory SSA-a unified approach for sparsely representing memory operations. In *Proc of the GCC Developers' Summit*. Citeseer.

Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 12–27.

Ghassan Shobaki, Maxim Shawabkeh, and Najm Eldeen Abu Rmaileh. 2013. Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 3 (2013), 14.

Vugranam C. Sreedhar. 1995. *Efficient Program Analysis Using DJ Graphs*. Ph.D. Dissertation. Montreal, Que., Canada, Canada. UMI Order No. GAXNN-08158.