A fast algorithm for global code motion of congruent instructions

Anonymous Author(s)

Abstract

We present a fast global-code-motion (GCM) compiler optimization which schedules congruent instructions across the program. Not only GCM saves code size, it exposes redundancies as well, it exposes more instruction level parallelism in the basic-block to which instructions are moved, and it enables other passes like loop invariant motion to remove redundancies. The cost model to drive the code motion is based on liveness analysis on SSA representation such that the (virtual) register pressure does not increase resulting in 2% fewer spills on the SPEC Cpu 2006 benchmark suite.

We have implemented the pass in LLVM. It is based on Global Value Numbering infrastructure available in LLVM. The experimental results show an average saving of 1% on the total compilation time on SPEC Cpu 2006. GCM enables more inlining and exposes more loop invariant code motion opportunities in the majority of benchmarks. We have also seen execution time improvements in a few of SPEC Cpu 2006 benchmarks viz. mcf and sjeng, moreover, register spills reduced by 2% on the SPEC Cpu 2006 benchmarks suite when compiled for x86_64-linux. GCM is an optimistic algorithm in the sense that it considers all congruent instructions in a function as potential candidates. We also formalize why register pressure reduces as a result of global-code-motion, and how global-code-motion increases instruction level parallelism thereby enabling more out-of-order execution on modern architectures.

Keywords Optimizing Compilers, GCC, LLVM, Code Generation, Global Scheduling

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 Common Subexpression Elimination (CSE) [Aho et al. 1986] work very well on single basic-blocks (local level) but when it comes to detect redundancies across basic-blocks (global level) these techniques fall short: more complex passes like Global Common Subexpression Elimination (GCSE) and Partial Redundancy Elimination (PRE) have been designed to handle these cases based on data-flow analysis [Morel and Renvoise 1979]. At first these techniques were described in the classical data-flow analysis framework, but later the use of Static Single Assignment (SSA) representation lowered their cost in terms of compilation time [Briggs and Cooper 1994; Chow et al. 1997; Kennedy et al. 1999a] and brought these techniques in the main stream: SSA based PRE is available in many compilers like GCC and LLVM.

This paper describes a fast algorithm for global-code-motion (GCM) of congruent instructions [Briggs et al. 1997], 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 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 similar to global code scheduling [Aho et al. 1986; Click 1995] in the sense that it will only move computations. Code hoisting, for example, can reduce the critical path length of execution in out-of-order machines. As more instructions are available at the hoisting point, the hardware has more independent instructions to reorder. The following reduced example illustrates how hoisting can improve performance by exposing more ILP.

```
float fun(float d, float min, float max, float a) {
  float tmin, tmax;
  float 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* depend 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: $C_{div} + 2(C_{sub} + C_{mul})$. For an out-of-order processor with two add units and two multiply units, the total latency of computation is $C_{div} + C_{sub} + C_{mul}$.

Conference'17, July 2017, Washington, DC, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

```
float fun(float d, float min, float max, float a) {
  float inv = 1.0f / d;
  float x = (min - a) * inv;
  float y = (max - a) * inv;
  float tmin = x, tmax = y;
  if (inv < 0) { tmin = y; tmax = x; }
  return tmax + tmin;
}
```

In this code the division operation can be executed in parallel with the two subtractions 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]. Although it decreases the instruction level parallelism (ILP) of basic blocks from where the instructions were removed, those basic blocks cannot execute for more cycles than the destination basic block. We have seen the performance of a proprietary benchmark (on an out-of-order processor) improve by 15%. In fact, there are several advantages of GCM:

- it helps reduce the code size of the program by replacing multiple instructions with one thereby making the function cheaper to inline because inliner heuristics depend on instruction count;
- it exposes more ILP 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 or sinking expressions out of branches, it can effectively reduce the amount of code to be speculatively executed;
- it reduces interference or register pressure with an appropriate cost-model: we have used a cost model in our implementation which only hoists or sinks when the register pressure is reduced;
- it reduces the total number of instructions to be executed for SIMD architectures which execute all code in branches based on masking or predication [Karrenberg and Hack 2011];
- it may improve loop vectorization by reducing a loop with control flow to a loop with a single basic-block, should all the instructions in a conditional get hoisted or sunk;
- it enables more loop invariant code motion (LICM): as LICM passes, in general, cannot effectively reason about instructions within conditional branches in the context of loops, code-motion is needed to move instructions out of conditional expressions and expose them to LICM.

There has been a lot of work both in industry and academia related to code hoisting/sinking, 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 integer linear programming (ILP) which results in prohibitively high compile time which is not suitable for industrial compilers. To the best of our knowledge we have not found any reference which explored global-code-motion with a cost-model to reduce register-pressure. A part of our implementation (aggressive code hoisting) is already merged in LLVM trunk. The main contributions of this paper are:

- an optimistic fast algorithm to schedule congruent instructions;
- introducing an analogue of ϕ nodes on the inverse CFG which allows faster tracking of anticipable instructions;
- a cost model to reduce interference and hence, reduce spills;
- experimental evaluation of our implementation in LLVM.

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 at 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 scalable.

Click [1995] describes aggressive global-code-motion in sparse representation. It first schedules 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 a speedup of as high as 23% but there is no cost model to limit amount of scheduling and could often introduce redundant computations. It is also not mentioned how safety is ensured while hoisting an instruction when it is not anticipable. The legality checks are only based on barriers (Section 4.1) and availability of used operands. Even introducing a simple integer addition can also result in undefined behavior [Wang et al. 2013]. Except from this paper we found no reference of global scheduling in sparse representation.

Dhamdhere [1988] and 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 data-flow model and an adaptation to a sparse SSA representation is required.

Rosen et al. [1988] also briefly discuss hoisting computations with identical value numbers from immediate successors. 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]. However, the cost-model to mitigate register pressure is missing, also, there is no mention of sinking congruent instructions.

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 (VBEs) 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 VBEs to a predecessor. It uses ANTIC_IN[B] to know what expressions will be computed on every path from the basic block 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_IN[B]. 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, working in tandem with PRE until no more hoisting is possible. 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.

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 X86, for example.

Shobaki et al. [2013] also presented a combinatorial global scheduler with reasonable performance improvements. It is possible that both Barany and Shobaki's implementation will have similar results when compiled for same target architecture. Also, both suffer from the same problem, although Shobaki not so much, of large compile times which would not suit in industrial compilers like GCC and LLVM. With the algorithm described in the next section, we got a reduction in register spills, improved inliner heuristics, improved compile time, and show performance improvements on SPEC Cpu 2006 benchmarks.

3 Global code motion

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

- Dominance (DOM) and Post-Dominance (PDOM) relations [Aho et al. 1986] on a Control Flow Graph (CFG);
- 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. 1989a];
- 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;
- Rank of an expression [Rosen et al. 1988]: The expressions are ordered according to the amount of dependency they have on other expressions. For example in the expression $a = b + c$, a will have higher rank than rank of b, and c.
- MemorySSA [Novillo 2007]: it is a factored use-def chain of memory operations that the compiler is unable to prove independent. MemorySSA accelerates the access to the alias analysis information.

Dominator tree, Global Value Numbers and Memory SSA are already available in LLVM but there is no facility to infer liveness of virtual registers. So we implemented DJ-Graph data structure that allowed us to calculate MergeSets which was used in the liveness analysis during GCM.

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

- find candidates (congruent instructions) suitable for global-code-motion (Section 4.4),
- compute a point in the program where it is both legal (Section 4.2) and profitable (Section 4.3) to move the code,
- move the code to hoist point or sink point (Section 4.5),
- update data structures to continue iterative global-code-motion (Section 4.5).

4 A fast algorithm for code motion congruent computations

Our algorithm factors out the reachability of values such that multiple queries to find reachability of values are fast. This is based on finding the dominance frontiers (DF) and the post-dominance frontiers (PDF) of the CFG. These points, being structural properties of a graph, do not change during code-motion. A post-dominance frontier is the basic block where anticipability (ANTIC) of an instruction (I) is checked because it is the basic block on which the basic block containing I is control dependent.

So we introduce a data structure (χ nodes) to keep track of values flowing out of a basic block with more than one successor. This approach allows us to hoist instructions to a basic block with more than two successors. The χ node

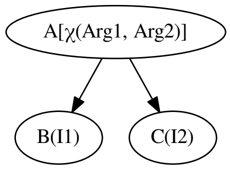

Figure 1. I1, I2 are instructions in B, and C respectively, Arg1 = {B, I1, V} and Arg2 = {C, I2, V}, V is the value number for both I1, and I2

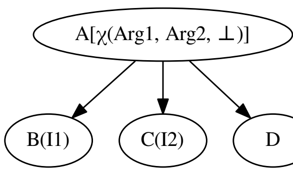

Figure 2. I1, I2 are instructions in B, and C respectively, Arg1 = {B, I1, V} and Arg2 = {C, I2, V}, V is the value number for both I1, and I2. A has missing entry in χ so V is not anticipable

in a basic block *B* has CHIArgs with three entries: a) the instruction (I) CHIArg refers to, b) the basic block *Dest* which is post-dominated by the basic block containing I, and c) the GVN of I.

```
struct CHIArg {
  // Edge destination (shows the direction of flow),
  // may not be where the I is.
  BasicBlock *Dest;
  // The instruction CHI-arg refers to.
  Instruction *I;
  // The global value number of I.
  GVNType VN;
};
```

A χ keeps information about 'congruent' values flowing out of a basic block. It is similar to ϕ [Cytron et al. 1989b] but in the inverse graph, and used for trackig out-flowing values on each edge. An illustration of χ is shown in Figure 1. The GVN for both I1 and I2 is V, the χ node will save the instruction as well as the edge where the value is flowing to. Where *Arg*1, *Arg*2 are of type CHIArg. The χ for both I1 and I2 is inserted in their PDF (A).

A similar idea has been used in SSAPRE [Kennedy et al. 1999b] for partial redundancy elimination in spare form. When SSAPRE inserts Φ (expression PHIs), it does so at places where (potentially) multiple expressions may merge. If a Φ has a missing entry, that means the expression is partially available. Subsequent algorithms work on Φs to find out redundancies. We extend the idea for GCM by propagating the anticipability in both directions.

For code hoisting we insert a χ, to a basic block with multiple successors. Then for each instruction, we register its flow of value to its corresponding χ which can be found in its PDF. Finally, we start walking the CFG and remove the χ if it has a missing entry (see Figure 2), that means the expression is not anticipable in the basic block having incomplete χ. For the remaining χs anticipabilty of 'values' is ensured by construction. For code-sinking we insert Φs in basic blocks with multiple predecessors. Then for each instruction we register its flow of value to its correponding Φ which can be found in its DF. Finally, we start walking the CFG and remove the Φ if it has a missing entry (see Figure 3), that means the expression is not available in the basic block having incomplete Φ. For the remaining Φs availability of 'values' is ensured by construction.

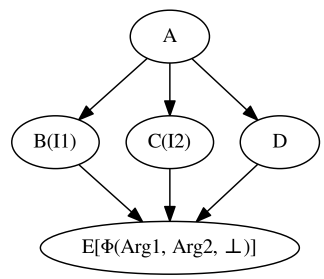

Figure 3. I1, I2 are instructions in B, and C respectively, Arg1 = {B, I1, V} and Arg2 = {C, I2, V}, V is the value number for both I1, and I2. A has missing entry in Φ so V is not available

In order to limit the number of spurious χs and Φs, [Park and Lee 2008], we insert them only for values with multiple occurrences in the function as they are the interesting candidates for this GCM. We believe this factorization of anticipability/availability of expressions may be suitable for faster implementation of other global (across basic blocks) analyses/optimizations as well. Before we consider factors affecting the legality of GCM, we introduce the concept of barriers.

4.1 Barriers

Barriers are based on the concept of pinned instructions [Click 1995] but extended to adapt to LLVM IR. A basic-block in LLVM IR is actually an extended basic-block because there might be non-returning calls in the middle of a basic-block. Essentially, any instruction that cannot guarantee progress is marked as a barrier. In the absence of context, as in our current implementation, some instructions which might be safe are still classified as barriers, e.g., calls with missing attributes.

```
// Compute barriers for both hoistable and sinkable
// candidates.
void computeBarriers()
```

```
for each basic-block B in a function:
  barrier_found = false
  for each instruction I in B:
    if I does not guarantee progress:
      mark I as a barrier instruction
      barrier_found = true
      break;

 // Find the last barrier below which instructions
 // can be sunk. If there was no barrier in B, any
 // instruction satisfying other legality checks
  // can be sunk.
  if barrier_found:
   for each instruction I in B in reverse order:
      if I does not guarantee progress:
        mark I as a sink barrier
        break
```

Computing barriers allows for efficient removal of non-movable candidates, this makes the legality check converge faster.

4.2 Legality of movable instructions

Since the equality of movable candidates is purely based on the value numbers, we also need to establish if hoisting them or sinking them would be legal. The completeness of χ and Φ are necessary for GCM but not sufficient. This is because congruence of value numbers only implies that they compute same value if the inputs are same. So for instructions like memory references, they may compute different values should the memory get modified along the path. For that we need to check for intersecting side-effects along the path.

For that, we need to prove that on all execution paths from the initial position of an instruction to its destination, the side effects appear in the same order. In general, we cannot introduce a new computation along any path of execution without checking for undefined behavior [Wang et al. 2013]. 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.

Scalars are the easiest to hoist because we only need to check for availability of operands. For sinking the scalars we need to make sure that there are no uses of the instruction along the path, For instructions with memory references like loads, stores and calls, Memory SSA allows efficient way of iterating over the use-def chains of memory references on a factored graph. Hoisting loads/stores across calls also requires precise analysis of all the memory addresses accessed by the call. The current implementation being an intraprocedural pass, cannot schedule aggressively across calls. In the presence of pure calls, loads can be hoisted but stores can't. Also, if a call throws exceptions, or if it may not return, nothing can be scheduled across that call.

4.3 Profitability check (Cost models)

After the legality checks have passed, we check if a GCM is profitable. That takes into account the impact GCM would have on various parameters that might affect runtime performance, e.g., impact on live-range, gain in the code size. The current implementation makes effort to not regress in performance but at the same time to reduce code size as much as possible. Sometimes, hoisting is not upward-safe [Click 1995], e.g., if the expressions are in a landing pad, sinking of those expressions may reduce code size. Following cost models are implemented:

4.3.1 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 GCM will decrease, 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. The following example explains how GCM of can reduce the register pressure.

```
B0: b = m
    c = n
    if p is true then goto B1 else goto B2
B1: a0 = b<kill> + c<kill>
    goto B3
B2: a1 = b<kill> + c<kill>
    goto B3
B3: ...
```

After hoisting, $a0$ and $a1$ are removed and a copy of $a0$ is placed in $B0$. Since b and c are killed in both $a0$ and $a1$, hoisting the expressions will reduce the register pressure because two registers will be freed at the insertion point but only one register will be required to assign the definition of hoisted instruction $a0$. We have reduced performance regressions with this cost model even if LLVM has a live-range splitting [Cooper and Simpson 1998] optimization.

Sinking may also reduce the register pressure in some cases, 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 fully 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: PHIc1, c2)
    PHI(d1, d2)
```

In this example (c1, c2) or (d1, d2) are potential sinkable candidates. Since (c1, c2) depend on i1 and i2 respectively which are also in their original basic blocks, c1 and c2 are not fully anticipable in B3. So without knowing the ability to sink 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.

4.3.2 Hoisting an expression across a 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 (also known as the caller saved registers) are saved because they might be modified by the callee and after the call they are restored. Due to this the register pressure is high because the number of available registers are reduced by the number of caller saved registers. In that situation a computation that increases register pressure is not profitable to hoist.

4.4 Finding candidates to schedule

The first step is to find a set of congruent instructions [Briggs et al. 1997]. This is performed by a linear scan of all instructions of the program and classifying them by their value numbers.

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 is to value number the address from where the value of a load is to be read from memory. 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.

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 4.5, we want to move as many instructions as possible, and it would require GVN values to be precomputed.

4.5 Code generation for the optimistic global-code-motion algorithm

Once all the legality and profitability checks are satisfied for a set of congruent instructions, they are suitable candidates for hoisting or sinking. A copy of the computation is inserted at the hoisting/sinking 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 updated as well.

After one iteration of algorithm runs through the entire function, it creates more opportunities 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 populate congruence classes as the program is modified by the code generation.

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.

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 as mentioned earlier in Section 4.4. We have implemented an optimistic global-code-motion of congruent instructions which uses the liveness analysis as illustrated by Das et al. [2012] and ranking expressions explained by Rosen et al. [1988].

GCM basically consists of two parts, i.e., hoisting and sinking. This implementation only moves congruent instructions. 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.

```
void doGCM()
  Analyses available:
    Dominator Tree, DFS Numbering, Memory SSA
  Compute DJ-graph of function
  constructMergeSet(CFG)
  computeBarriers()
```

```
do:
  Compute GVN of each expression in the function
 // Repeat hoisting if any candidates were hoisted.
while (doCodeHoisting() > 0)
doCodeSinking()
```

We collect the GVN of all the instructions in the function and iterate on the list of instructions having identical GVNs. The algorithm *doGCM* prefers hoisting (*doCodeHoisting*) to sinking (*doCodeSinking*).

```
int doCodeHoisting(VNs)
  // VNs = map (VN -> list of I with value VN)
  Sort VNs in ascending order

  for each VN in VNs
    for each instruction with the same VN
      Find its post-dominance-frontier (PDF)
      Insert the CHI-node at PDF

  for each CHI-nodes in the CFG:
    Remove the ones with missing entries
    Remove CHI-arg if it fails legality checks

  for each CHI-nodes in the CFG:
    if CHI  does not have missing entries:
      Perform code hoisting

  return number of sinked instructions
```

Code hoisting opens new opportunities for other hoistable candidates which were of higher rank (depended on candidates which got hoisted) which are encountered subsequently as the algorithm visits instructions in the increasing order of their ranks.

After no more candidates are hoistable, sinking is performed. Sinking is only performed once because currently there are very few sinkable candidates (Figure 4) as SimplifyCFG of LLVM (which runs before GCM) has a code-sinking which already sinks several instructions.

For sinking we reuse the ϕ nodes instead of inserting another expression nodes like in [Kennedy et al. 1999b] because ϕ nodes ensure availability of definitions, and hence downward safety, as long as all instructions corresponding to a ϕ are congruent. Also, the ϕ nodes represent a complete set of instructions which could be potentially sunk for code-size optimizations because ϕ nodes represent all values flowing out from a basic block into its dominance-frontier.

```
int doCodeSinking()
  // Find sinkable instructions
  For each value number VN > 1 instructions:
   if there are two instructions with same immediate
        successor(S) as post-dominator:
     if both are only used in the same $\phi$-node of S
        && dependencies of both are available at S:
         mark instructions as sinkable

  // Sink the instructions and update statistics
  For each pair (I1, I2) of sinkable instructions:
    Move I1 to the sink point
    // the sink point is just after all
    // the PHI-nodes in the post-dominator
    update all the uses of PHI node (PN) with I1
    Remove I2 and PN
    if I1 is a memory reference:
      update MemorySSA of I1
      update MemorySSA of removed instructions
        to point to the MemorySSA reference of I1
      remove MemorySSA reference of all
        others which were deleted
    update statistics

  return number of sinked instructions
```

4.6 Time complexity of algorithm

The complexity of code hoisting is linear in number of instructions that are candidates for global-code-motion, matching the complexity of PRE on SSA form. The analyses computed for this pass like Global Value Numbering, Marking Barriers, are both linear in number of instructions in a function. Computation of DJ-graph and merge-sets is $O(E)$, E being number of edges in the CFG. Liveness analysis is not very expensive [Das et al. 2012] but still only performed on-demand for hoistable candidates. Other analyses like Alias Analysis, MemorySSA and Dominator Tree are already available in the LLVM pass pipeline.

Although we recompute GVN, and Barriers for each iteration of the global-code-motion, there is still gain in the compilation times (see Section 3). We have also provided appropriate compiler flags to expedite global-code-motion by bailing out with fewer iterations, or skip the liveness based profitability analysis to aggressively move the code as long as they are legal.

5 Experimental Evaluation

For evaluation of global-code-motion, we built SPEC Cpu 2006 with the patch. All the experiments were conducted on an x86_64-linux machine and at -Ofast optimization level. Each benchmark was run three times and the best result was taken. We collected execution time and code-size which is listed in Table 1, other compiler statistics are listed in Table 2, Table 4, and Figure 4. We also measured compile time Table 3 to see the impact of global-code-motion.

Overall there was only a minor change in the performance numbers with a few improvements (429.mcf, 458.sjeng, etc.) and a few regressions (447.dealII, 453.povray, etc.) The global-code-motion pass was run twice in the pass pipeline (at -Ofast), first after EarlyCSE pass, and then after the inliner. Because EarlyCSE removes local redundancies, GCM would

Benchmarks	% performance uplift at -Ofast (high better)	size reduction (Bytes) at -Ofast (high better)	size reduction (Bytes) at -Os (high better)
400.perlbench	1.00	-9120	544
401.bzip2	1.00	-1600	128
403.gcc	1.00	-58176	3384
429.mcf	1.03	160	16
433.milc	1.00	-656	224
444.namd	1.00	-3864	-1168
445.gobmk	0.99	-20480	-264
447.dealII	0.95	-27836	50934
450.soplex	0.99	2220	348
453.povray	0.97	164	-472
456.hmmer	0.99	-3464	-352
458.sjeng	1.02	440	1456
462.libquantum	1.00	-96	0
464.h264ref	1.00	-3424	864
470.lbm	1.00	0	0
471.omnetpp	1.00	2688	200
473.astar	1.00	-928	152
482.sphinx3	0.99	3008	-176
483.xalancbmk	1.01	46492	4012
Geomean	0.997		

Table 1. Execution time (ratio) and code size change (Bytes) with and without GCM on SPEC Cpu 2006: performance uplift at in mcf and sjeng; code size improvement in dealII.

have to analyze less redundant instructions, also as inliner creates more redundancies, GCM would have more opportunities to move congruent instructions out of branches.

We see some nice gains in code size at -Os with global-code-motion (403.gcc, 447.dealII, and 483.xalancbmk) as shown in Table 1. The code size listed here is the text size delta (from linux size command) of the final executable for each benchmark. On the other hand, some benchmarks like 473.astar increased in code size by almost 2%. This is because once the code-size of a function decreases (due to GCM) it becomes cheaper to inline. As we can see in Table 4, four more functions got inlined in 473.astar. Except for 447.dealII, 450.soplex, 482.sphinx3 and 483.xalancbmk all other benchmarks got more functions inlined. Also, since the pass runs early, it affects many optimizations which rely on the number of instructions, length of the use-def chain, and other metrics.

The code-size at -Ofast is very widespread with huge gains in 483.xalancbmk to serious regressions in 403.gcc. This is mostly because the compiler's focus is to improve performance at the cost of code-size at -Ofast, so passes like inlining, loop-unrolling, code-versioning are aggressive at -Ofast.

Spills are listed in Table 2. Spills were reduced by an average of 2% on the SPEC Cpu 2006 benchmark suite. This is a result of cost-model which limits the global-code-motion to those candidates which do not increase the register pressure (except for loads Section 4.3) overall spills still decreases.

The compilation time actually reduced for most of the benchmarks as a result of code motion. This is because, as global-code-motion removes (congruent) instructions, the number of instructions to be processed by later compilation passes also reduces. The compilation time listed here is the total instruction count executed during the compilation of each benchmark as output from valgrind --tool=cachegrind [Nethercote and Seward 2007].

Benchmarks	base(-Ofast)	GCM(-Ofast)	GCM/base
400.perlbench	2539	2493	0.98
401.bzip2	718	707	0.98
403.gcc	5782	5722	0.99
429.mcf	14	17	1.21
433.milc	635	642	1.01
444.namd	3223	3274	1.01
445.gobmk	2166	2173	1.00
447.dealII	11074	10234	0.92
450.soplex	1122	1125	1.00
453.povray	4701	4692	1.09
456.hmmer	1197	1274	1.06
458.sjeng	168	176	1.05
462.libquantum	118	118	1.00
464.h264ref	3379	3454	1.02
470.lbm	33	33	1.00
471.omnetpp	524	527	1.00
473.astar	180	201	1.12
482.sphinx3	674	670	0.99
483.xalancbmk	5102	4965	0.97
Grand Total	43349	42497	0.98

Table 2. Number of spills before and after global-code-motion(GCM) on SPEC2006 at -Ofast. Lower is better in %age change. Spills reduced by 2%.

Benchmarks	Baseline (in Millions)	GCM (in Millions)	%Decrease (lower is better)
400.perlbench	186	184	0.99
401.bzip2	76	75	0.99
403.gcc	1,079	1,073	0.99
429.mcf	78	77	0.99
433.milc	272	269	0.99
444.namd	78	77	0.99
445.gobmk	261	258	0.99
447.dealII	643	638	0.99
450.soplex	245	242	0.99
453.povray	499	494	0.99
456.hmmer	214	211	0.99
458.sjeng	89	88	0.99
462.libquantum	84	84	0.99
464.h264ref	151	149	0.99
470.lbm	71	71	1.00
471.omnetpp	382	379	0.99
473.astar	78	77	0.99
482.sphinx3	158	156	0.99
483.xalancbmk	22,485	22,455	1.00

Table 3. Change in total number of instructions executed during the compilation with and without global-code-motion (GCM) on SPEC Cpu 2006 at -Ofast. The instructions were counted using valgrind.

Some useful static metrics are presented in Table 4, which shows how global-code-motion impacted important compiler optimizations like inlining and LICM. LICM removes loop invariant code out of the loop. It may hoist the code to the loop's preheader or sink the code to loop's post-dominator [Muchnick 1997]. As we can see number of instructions hoisted increased significantly in several benchmarks, although number of instructions sunk did not change by much.

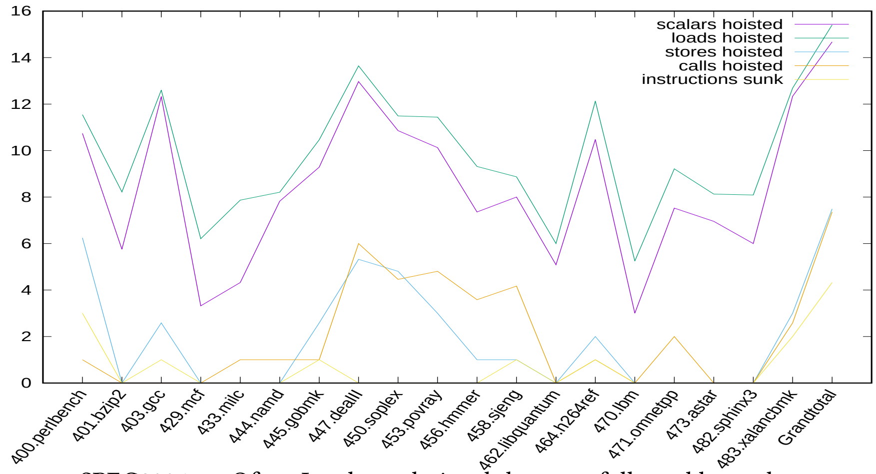

Figure 4. GCM stats on SPEC2006 at -Ofast. Loads are hoisted the most followed by scalars, stores and calls.

	Loop Invariant Motion		Functions	
Benchmarks	Hoisted %	Sunk %	Inlined%	Deleted%
400.perlbench	2.1	-2.8	0.2	0.9
401.bzip2	14.4	0.0	4.7	3.1
403.gcc	13.7	27.3	0.9	0.1
429.mcf	-7.3	0.0	0.0	0.0
433.milc	10.2	0.0	8.0	0.0
444.namd	0.1	0.0	0.8	-3.5
445.gobmk	0.6	5.6	5.4	1.7
447.dealII	62.2	-86.7	-0.2	-0.8
450.soplex	3.9	-5.6	-0.2	-0.4
453.povray	-4.8	39.1	0.2	-0.1
456.hmmer	1.0	166.7	0.0	-1.9
458.sjeng	11.5	0.0	6.3	0.0
462.libquantum	6.6	0.0	0.0	0.0
464.h264ref	0.5	0.0	13.0	0.0
470.lbm	0.0	0.0	0.0	0.0
471.omnetpp	17.0	-6.3	1.1	-0.4
473.astar	9.1	0.0	1.1	0.0
482.sphinx3	5.9	-18.2	-2.0	0.0
483.xalancbmk	14.6	-3.6	-1.0	-1.2

Table 4. Change in the static compile time metrics w.r.t. global-code-motion (GCM) on SPEC Cpu 2006 at -Ofast. The % columns show percentage increase in metric w.r.t. baseline. GCM improves LICM and inlining in general.

Number of inlined functions also improved for most benchmarks. In general when more functions were inlined, more functions were deleted as in 400.perlbench, 445.gobmk.

The compile time metrics of GCM are listed in Figure 4. 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 code-motion 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 pass which scalarizes some memory references) in the LLVM pass pipeline. Passes like mem2reg and instcombine might actually remove those loads and the number of hoisted loads may change should the GCM pass be scheduled later. We can see a few sunk instructions even if LLVM has a code sinking and hoisting transforms in the SimplifyCFG pass which runs before GCM: this is because sinking and hoisting in SimplifyCFG have some rather severe limitations to make the analysis and code transform very fast in order to be able to run the pass several times. Another reason is that instructions for which dependencies are not directly available in the successor (Section 4.2) is not sunk for now because that would require a look ahead on the ability to sink the dependencies. We plan to implement this in near future.

6 Conclusions and Future Work

We have presented the global-code-motion of congruent computations in SSA form. We saw that it improves inlining and LICM opportunities, reduces register spills, it also improves performance in some cases. To preserve performance and not hoist/sink too much, we have implemented a register pressure aware cost model as described in Section 4.3. A part of our implementation is merged in LLVM trunk as GVNHoist.cpp and the GCM implementation is under review.

This pass does not hoist fully redundant expressions because they are already handled by GVN-PRE. It is advisable to run gvn-hoist before and after gvn-pre because gvn-pre creates opportunities for more instructions to be hoisted.

We would like to integrate GCM with LLVM's new GVN-PRE implementation such that more congruent instructions can be scheduled. This would also expose more redundancies to the GVN-PRE. In general it is a good idea to start with lower ranked expressions 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 compile time regression have been observed. This can be improved by ranking the computations [Rosen et al. 1988]. Also, since GCM runs very early in the pass pipeline, it will be good to evaluate the code size/performance impact when it is run in sync with GVN-PRE just like GCC does. With the implementation of GCM in LLVM, the passes which rely on the code-size or instruction-counts to make optimization decisions need to be revisited for example, the inliner.

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.

Preston Briggs, Keith D Cooper, and L Taylor Simpson. 1997. Value numbering. *Software Practice & Experience* 27, 6 (1997), 701–724.

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.

Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989a. 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.

Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989b. 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.

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: 5738, 11820, 11832, 21485, 23286, 29144, 32590, 33315, 35303, 38204, 43159, 52256. (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

Ralf Karrenberg and Sebastian Hack. 2011. Whole-function Vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11).* IEEE Computer Society, Washington, DC, USA, 141–150. http://dl.acm.org/citation.cfm?id=2190025.2190061

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

Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. 1999b. 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.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

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

Jong-Soo Park and Jae-Jin Lee. 2008. A Practical Improvement to the Partial Redundancy Elimination in SSA Form. *Journal of Computing Science and Engineering* 2, 3 (2008), 301–320.

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.

Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* ACM, 260–275.