

A fast global code motion algorithm for code size optimization

ANONYMOUS AUTHOR(S)

We present a global-code-motion (GCM) compiler optimization which schedules congruent instructions across the program. Not only GCM saves code size, it exposes redundancies in some cases, 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 more 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 majority of the 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 make an extra effort to hoist candidates by partitioning the potential candidates in a way to enable partial hoisting in case a common hoisting point for all the candidates cannot be found. 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.

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

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

ACM Reference format:

Anonymous Author(s). 2017. A fast global code motion algorithm for code size optimization. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), ?? pages.
DOI: 10.1145/nnnnnnnn.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 Common Subexpression Elimination (CSE) (?) 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 (?). 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 (???) and brought these techniques in the main stream: nowadays SSA based PRE is available in industrial compilers like GCC and LLVM.

This paper describes global-code-motion (GCM) of congruent instructions (?), a technique that uses the information computed for PRE to detect identical computations but has a transformation phase whose

A note.

2017. 2475-1421/2017/1-ART1 \$15.00

DOI: 10.1145/nnnnnnnn.nnnnnnn

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 (??) 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 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 $C_{div} + C_{sub} + C_{mul}$.

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 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} (??). We have seen the performance of an out-of-order processor going up by 15% on a benchmark containing the above motivating example running in a tight loop. In fact, there are several advantages of GCM:

- it helps reduce the code size of the program by replacing multiple instructions with one although the final code size may be higher because of improved inlining heuristics; functions become cheaper to inline when their code size decreases because the inliner heuristics depend on instruction count;
- it exposes 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 or 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 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;
- 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 to hoist and sink code out of branches, and in general global scheduling (?). Some relate code-hoisting to code-size optimization (?) and many (??) 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. To the best of our knowledge we have not found any reference which explored global-code-motion of congruent instructions in as much detail as us. A part of our implementation (aggressive code hoisting) is already merged in LLVM trunk, however, a general implementation of global-code-motion of congruent (non redundant) expressions is still missing from GCC and LLVM trunk. The main contributions of this paper are:

- a new optimistic algorithm to move congruent instructions out of branches;
- a cost model to reduce interference and hence, 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 register spills and improve performance in some cases.

2 RELATED WORK

There are a lot of bug reports in GCC and LLVM bugzillas (??), 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.

? and ? 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.

? 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 (?). However, the cost-model to mitigate register pressure is missing, also, there is no mention of sinking congruent instructions.

GCC recently got code-hoisting (?) 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.

? describes 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 a speedup of as high as 23%.

? 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.

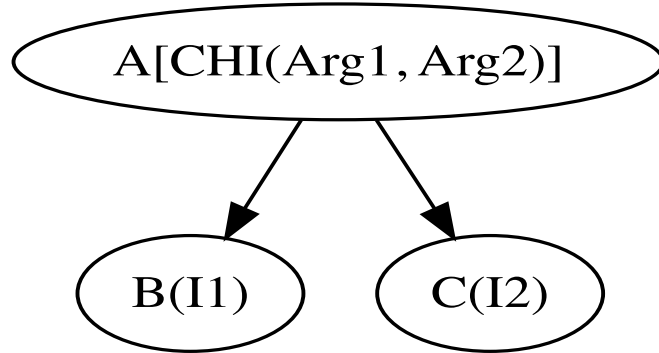
? also presented a combinatorial global scheduler with reasonable performance improvements. It is possible that both Barany and Shobaki's implementation will have similar results when compiled for same target architecture. Also, both suffer from the same problem, although Shobaki not so much, of large 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 CODE HOISTING OF CONGRUENT COMPUTATIONS

Examples to show CHI insertion.

The algorithm factors out the reachability of values such that multiple queries to find reachability of values are fast. This is based on finding the ANTIC points in the CFG which do not change during hoisting. The ANTIC points are basically the dominance-frontiers in the inverse graph. So we introduce a data structure (CHI nodes) to keep track of values flowing out of a basic block. We only do this for values with multiple occurrences in the function as they are the potentially hoistable candidates. This approach allows us to hoist instructions to a basic block with more than two successors, as well as deal with infinite loops in a trivial way.

Limitations: 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.



A CHI keeps information about values flowing out of a basic block. It is similar to PHI but in the inverse graph, and used for outgoing values on each edge. For conciseness, it is computed only for instructions with multiple occurrences in the CFG because they are the only hoistable candidates.

The global value number for both I1 and I2 is V, the CHI node will save the instruction as well as the edge where the value is flowing to. Where CHIArg1, CHIArg2 are of type CHIArg.

```

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

Essentially, CHIArg1 = V, B, I1 and CHIArg2 = V, C, I2

For sinking we use the similar approach as ?? to track values flowing into a basic block.

TODO: Explain ranking

4 GLOBAL CODE MOTION

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

- Dominance (DOM) and Post-Dominance (PDOM) relations (?) on a Control Flow Graph (CFG);
- DJ-Graph (?) 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 (?);
- Static Single Assignment (SSA) (?);

- Global Value Numbering (GVN) (??): 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;
- MemorySSA (?): 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.

Dominance graphs, Global Value Numbers and Memory SSA are already available in LLVM but there is no facility to infer liveness of virtual registers. For that we implemented DJ-Graph data structure that allows us to calculate MergeSets which is used in the liveness analysis in our implementation of GCM.

4.1 Liveness using DJ-graph

DJ-graph (?) 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 (?). 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 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 the merge-set computation based on a DJ-graph as illustrated in (?). A simplified version of the 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 MergeSet)
```

```
    for e in (all incoming edges to lnode)
```

```
        if e is in JE and e in V
```

```
            if MergeSet(Source Node of e) is not a subset of MergeSet(lnode)
```

```
                return true
```

```
    return false
```

Each basic block which has a join-edge (J-Edge) to its successor will have a merge-set that is a subset of the merge-set of its successor (?). The function *still_inconsistent* returns true if this invariant is not satisfied which allows the function *constructMergeSet* to repeat.

```
// Compute a merge-set top-down in breadth first order.
```

```
bool constructMergeSet_1(BFSList B, JEdges JE, DomLevel DL)
```

```
    repeat = false
```

```
    V = empty list of visited edges
```

```
    for n in B
```

```
        for e in (all incoming edges to n)
```

```
            if e is in JE and e not in V
```

```
                V(e) = true
```

```
                let tmp = Source Node of e
```

```
                let tnode = Target Node of e
```

```
                let lnode = NULL
```

```
                while (DL(tmp) >= DL(tnode))
```

```
                    MergeSet(tmp) = MergeSet(tmp) U MergeSet(tnode) U {tnode}
```

```
                    lnode = tmp
```

```
                    tmp = dom-parent(tmp)
```

```
1         repeat = still_inconsistent(lnode, JE, MergeSet)
2     return repeat
```

3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

```

1 // Construct merge-set of each node in Control-Flow-Graph G of a function.
2 void constructMergeSet(CFG G)
3     B = Breadth First Order of G
4     JE = JEdges of G
5     DL = List of Path length (from root) of each node in G.
6     do // Call until a fixed point is reached.
7         Repeat = constructMergeSet_1(B, JE, DL)
8     while (Repeat)
9
10    We compute merge sets, of the control flow graph, which remain same throughout the GCM trans-
11    formation. For GCM we only want to know if a use operand is a kill (to compute changes in register
12    pressure). For that we only need to know whether the use is also required later in the execution path.
13    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
14    relation on-demand when profitability of hoist/sink candidates is to be evaluated. A simplified version of
15    isLiveOutUsingMergeSet is presented here:
16    // Return true if variable A is liveout from basic-block N
17    bool isLiveOutUsingMergeSet(Node N, Variable A)
18        if (A is defined in N)
19            if (A is used outside any basic-block other than N)
20                return true
21            return false
22
23    MergeSet(N) = null
24    // The merge-set of N is the union of merge-sets of its successors.
25    for W in successors(N)
26        MergeSet(N) = MergeSet(N) U MergeSet(W)
27
28    // Iterate over all the uses of A and see if any intersect with the merge-set of N.
29    for T in users(A)
30        B = basic_block(T)
31        while (B != null and B != def_bb(A))
32            if B in MergeSet(N) return true
33            B = dom-parent(B)
34        return false
35
36    If a variable is not live-out of a basic-block, it still may be used later in the basic-block. To establish
37    whether a use of a variable is a kill we iterate on all the subsequent instructions in a basic-block checking
38    for uses, the isKill function uses isLiveOutUsingMergeSet to find out if an operand is killed in an
39    instruction.
40    // Return true if op is the last use of the operand in I.
41    bool isKill(Operand op, Instruction I)
42        if (isLiveOutUsingMergeSet(op))
43            return false
44        for each I1 after I in basic block of I
45            if (I1 uses op)
46                return false // Not a kill
47        return true

```


The original algorithm presented in (?) 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.

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 ??),
- compute a point in the program where it is both legal (Section ??) and profitable (Section ??) to move the code,
- move the code to hoist point or sink point (Section ??) and
- update data structures to continue iterative global-code-motion (Section ??).

4.2 Finding candidates to move

The first step is to find a set of congruent instructions (?). This is performed by a linear scan of all instructions of the program and classifying them by their value numbers. 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 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. 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 ??, 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 congruence classes, we compute for each group of congruent instructions, a point in the program that is both legal and profitable for the instructions to be moved to.

4.3 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 to reinstantiate or rematerialize 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 moved 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.

4.3.1 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.

4.3.2 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 address 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 the candidate is discarded. To iterate on the use-def chains for memory references the MemorySSA infrastructure of LLVM is used.

4.3.3 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, and that there are no aliasing loads or store along the path from the current position to the hoisting point.

4.3.4 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.

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 a call throws exceptions, or if it may not return, nothing can be hoisted across that call.

4.3.5 Legality of sinking expressions. Sometimes, hoisting is not upward-safe (?), e.g., if the expressions are in a landing pad, sinking of those expressions may reduce code size. 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: phi(c1, 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.

A general global scheduling algorithm also requires checks for undefined values 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 instructions. This simplifies the implementation.

4.3.6 Barriers. Barriers are based on the concept of pinned instructions (?) but extended to adapt to LLVM IR. Since a basic-block in LLVM IR is actually an extended basic-block because there might be calls in the middle of a basic-block which might not return. 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., volatile loads/stores, calls with missing attributes.

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

4.3.7 Downward-safety of movable instructions. In order to be able to move an instruction that has side effects, we must guarantee that the side-effects of that instruction before and after its code-motion will produce the same side-effects. For hoisting in particular, we need to prove that on all execution paths from the hoisting point to the end of the program 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 behaviors. The algorithm *compute_downward_safety* computes if hoisting a set of instruction to a common hoisting point is downward-safe (?). Since GCM is only dealing with VBEs, computing downward-safety ensures that no undefined behaviors are introduced as a result of code-motion.

```
// Return true if a hoistable instruction is downward safe at hoist-point
```

```

1 bool compute_downward_safety()
2   Worklist = list of all basic-blocks of hoistable instructions
3   for each basic-block B in the dominator tree starting at hoist-point:
4     if Worklist is empty
5       return false // Path exists! not downward safe
6     if B is in Worklist:
7       remove B from worklist
8       remove subtree with root at B // Available at B => downward safe from B
9     if B is a leaf basic-block
10      return false // Path exists! not downward safe
11     if B dominates hoist-point
12      return false // Back edge
13   return true

```

4.4 Profitability check (Cost models)

After the legality checks have passed, we check if a global-code-motion is profitable. That takes into account the impact global-code-motion 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 across popular benchmarks and at the same time to reduce code size as much as possible. Following are the cost models which are implemented:

4.4.1 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 global-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. The following example explains how global-code-motion of identical computations can reduce the register pressure.

```

29 B0: b = m
30     c = n
31     if c is true then goto B1 else goto B2

```

```

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

```

```

35 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. Since `'b'` and `'c'` are killed in `'a0'` and `'a1'`, hoisting the expressions will reduce the register pressure because two registers will be freed.

Ideally, it should be okay to hoist all the instructions and a later live-range-splitting (?) 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 optimal decision and we have found performance 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 corresponding loads/stores would not be legal to hoist. In order to reduce the register pressure while hoisting loads, we have restricted hoisting of address computations away from their corresponding loads and stores when

```

The file abc.ps hasn't been created from abc.dot yet.
Run 'dot -Tps -o abc.ps abc.dot' to create it.
Or invoke LATEX with the -shell-escape option to have this done automatically.

```

Fig. 1. CFG to illustrate partitioning

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 these regressions.

There is a special case for hoisting load instructions when the hoist-point is the predecessor basic-block for all the congruent loads. Even if the register pressure would increase, we prefer to hoist loads in this case. The reason being, that would make the loaded value available by the time it is used. Also, because stores and calls are hoisted the least, see Table ??, the performance does not change much whether they are hoisted or not.

4.4.2 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. 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 a computation that increases register pressure is not profitable to hoist.

4.4.3 Partitioning the list of hoist candidates to maximize hoisting. In the approach, described in Section ??, it is possible that a common hoisting point for all the instructions does not satisfy legality or profitability checks. 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 hoist instructions. By sorting the list of all the candidates in the increasing order of their depth first search discovery time stamp (?) (DFSIn numbers,) we make sure that candidates closer in the list have their common dominator nearby. In Figure-??, if B3, B4, B5, and B6 have identical computations and if for some reason, they cannot be hoisted 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. Even if instructions in (B3, B4) could not be hoisted, for some reason, to B1, the instructions in (B5, B6) can still be hoisted – if legal and profitable – to B2.

In our current implementation we keep as many candidates in one set as possible (greedy approach.) To start with, first two candidates are tried for hoistability (see Section ??,) if that passes then the next instruction is tried (if available) and so on until a non-hoistable instruction is found. The partition starts from that point. 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.

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 also updated to get up-to-date information about memory references.

After one iteration of algorithm runs through the entire function, it creates more opportunities for *higher ranked* computations (?). 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 (?), 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 ???. We have implemented an optimistic global-code-motion of congruent instructions which uses the liveness analysis as illustrated by ? and ranking expressions explained by ?.

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
    compute mergeset based on that (using constructMergeSet function)
    compute barriers (using compute_barriers function)
    do
        Compute GVN of each expression in the function
    while (doCodeHoisting() > 0) // Repeat hoisting if any candidates were hoisted.
    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*).

For hoisting, we find a common dominator dominating all a set of congruent instructions and perform legality checks, as described in Section ???. 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 algorithm would partition (Section ??) the list of identical instructions into subsets which can be partially hoisted to their respective common dominators.

```

1  int doCodeHoisting() // Find hoistable candidates and do hoisting
2      For each value number VN with 2 or more instructions (I1, I2, ..., Ik)
3          sort the instructions according to DFSIn numbers
4          if first two I1, I2 are hoistable and if they are downward-safe at hoist-point
5              and if they are profitable to be hoisted at hoist-point
6              proceed to check the ability to hoist subsequent instruction with same VN
7          else
8              proceed to check if I2 and I3 (if available) can be hoisted
9      For each set of instructions (I1, ..., Ik) with same VN that can be hoisted:
10         move the first one I1 to the hoist-point
11         update the use of all other candidates (I2, ..., Ik) to refer to I1
12         remove all other instructions (I2, ..., Ik)
13         if I1 has memory references (a load, store, call, etc.)
14             update MemorySSA of I1 and others to point to the MemorySSA reference of I1
15             remove MemorySSA reference of all others which were deleted
16         update statistics
17     return number of hoisted instructions

```

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

```

21 void doCodeSinking()
22     // Find sinkable instructions
23     For each value number VN with 2 or more instructions
24         if there are two instructions with same immediate successor as post-dominator
25             if both are only used in the same PHI-node (PN) of the successor
26                 if both have dependencies that are available at the sink point
27                     // available or can be made available by rematerialization
28                     mark VN as sinkable
29
30     // Sink the instructions and update statistics
31     For each pair (I1, I2) of sinkable instructions
32         Move I1 to the sink point
33         // the sink point is just after all the PHI-nodes in the post-dominator
34         update all the uses of PHI node (PN) with I1
35         Remove I2 and PN
36         if I1 is a memory reference:
37             update MemorySSA of I1
38             update MemorySSA of removed instructions to point to the MemorySSA reference of I1
39             remove MemorySSA reference of all others which were deleted
40         update statistics
41
42     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). Ideally we could iterate on lower ranking expressions first and then proceed to higher ranking expressions in the same iteration but LLVM's GVN infrastructure

does not compute equivalence classes in an effective way. We found it simpler to just recompute the value numbers and start finding hoistable candidates again.

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 are Global Value Numbering, Computation of DJ-graph and merge-sets, Marking Barriers, all are linear in number of instructions in a function. Liveness analysis is expensive and only performed on-demand for hoistable candidates so it does not affect the overall compilation time by much. 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, we have not seen significant increase in the compilation times (see Section ??). 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.

The analysis is followed by a simple code generation that adds the identified instruction in the destination point and removes all instructions rendered redundant as a result of global-code-motion.

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 ??, other compiler statistics are listed in Table ??, Table ??, and Table ?. We also measured compile time Table ?? to see the impact of global-code-motion.

Benchmarks	% performance uplift at -Ofast (high better)	size reduction in Bytes at -Ofast (high better)	size reduction in 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.hmmmer	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 global-code-motion (GCM) on SPEC Cpu 2006: performance uplift at -Ofast in mcf and sjeng; code size improvement in dealII at -Os.

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 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 ???. 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 ??, 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 very aggressive at -Ofast.

Number of spills	base(-Ofast)	GCM(-Ofast)	%change (low better)
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.hmmmer	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. Spills reduced by 2%.

Spills are listed in Table ???. 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 ??,) 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 (?).

Some useful static metrics are presented in Table ??, 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 (?). As we can see number of instructions hoisted increased significantly in several benchmarks, although number of instructions sunk did not change by much. Number of inlined functions also improved for

Benchmarks	Baseline	GCM	%Decrease (low better)
400.perlbench	186,829,426	184,690,006	0.99
401.bzip2	76,337,033	75,942,641	0.99
403.gcc	1,079,625,231	1,073,158,220	0.99
429.mcf	78,336,298	77,859,513	0.99
433.milc	272,525,306	269,684,876	0.99
444.namd	78,354,313	77,875,603	0.99
445.gobmk	261,229,078	258,459,798	0.99
447.dealII	643,457,797	638,628,331	0.99
450.soplex	245,442,755	242,811,500	0.99
453.povray	499,034,715	494,785,035	0.99
456.hmmer	214,425,324	211,984,389	0.99
458.sjeng	89,612,524	88,803,166	0.99
462.libquantum	84,762,158	84,077,905	0.99
464.h264ref	151,049,588	149,284,624	0.99
470.lbm	71,878,726	71,775,405	1.00
471.omnetpp	382,661,268	379,123,833	0.99
473.astar	78,351,117	77,872,820	0.99
482.sphinx3	158,434,290	156,586,406	0.99
483.xalancbmk	22,485,708,759	22,455,060,569	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.

Benchmarks	Hoisted by LICM			Sunk by LICM			Functions Inlined			Functions Deleted		
	Base	GCM	%	Base	GCM	%	Base	GCM	%	Base	GCM	%
400.perlbench	2428	2480	2.1	36	35	-2.8	1562	1565	0.2	232	234	0.9
401.bzip2	721	825	14.4	3	3	0.0	172	180	4.7	32	33	3.1
403.gcc	8867	10079	13.7	143	182	27.3	5034	5080	0.9	1303	130	0.1
429.mcf	55	51	-7.3	1	1	0.0	4	4	0.0	0	0	0.0
433.milc	2537	2795	10.2	3	3	0.0	75	81	8.0	0	0	0.0
444.namd	2204	2206	0.1	240	240	0.0	387	390	0.8	57	55	-3.5
445.gobmk	3090	3107	0.6	18	19	5.6	735	775	5.4	230	234	1.7
447.dealII	59114	95884	62.2	60	8	-86.7	100891	100692	-0.2	30219	29966	-0.8
450.soplex	4668	4848	3.9	18	17	-5.6	9801	9784	-0.2	3125	3112	-0.4
453.povray	9533	9079	-4.8	23	32	39.1	5522	5532	0.2	1202	1201	-0.1
456.hmmer	3200	3232	1.0	3	8	166.7	562	562	0.0	103	101	-1.9
458.sjeng	400	446	11.5	0	0	0.0	127	135	6.3	18	18	0.0
462.libquantum	362	386	6.6	0	0	0.0	87	87	0.0	21	21	0.0
464.h264ref	17075	17153	0.5	4	4	0.0	338	382	13.0	72	72	0.0
470.lbm	39	39	0.0	0	0	0.0	13	13	0.0	4	4	0.0
471.omnetpp	565	661	17.0	32	30	-6.3	4790	4843	1.1	1321	1316	-0.4
473.astar	691	754	9.1	0	0	0.0	359	363	1.1	113	113	0.0
482.sphinx3	1599	1693	5.9	11	9	-18.2	198	194	-2.0	53	53	0.0
483.xalancbmk	9997	11453	14.6	56	54	-3.6	69534	68839	-1.0	29875	29519	-1.2

Table 4. Static metrics before (Base) and after global-code-motion (GCM) on SPEC Cpu 2006 at -Ofast. The % Column show percentage increase in metric w.r.t. Base. GCM improves LICM and inlining.

most benchmarks. In general when more functions were inlined, more functions were deleted as in 400.perlbench, 445.gobmk.

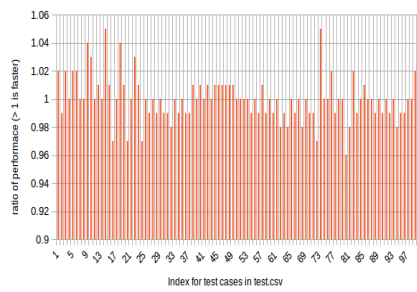
The compile time metrics of GCM are listed in Table ???. The table lists the number of scalars, loads, stores and calls hoisted as well as removed. For each category, the number of instructions removed is greater or equal to the number of instructions hoisted because each code-motion is performed only 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 ??) is not sunk for now. Sinking those instructions would require a look ahead on the ability to sink the dependencies. We plan to implement this in near future.

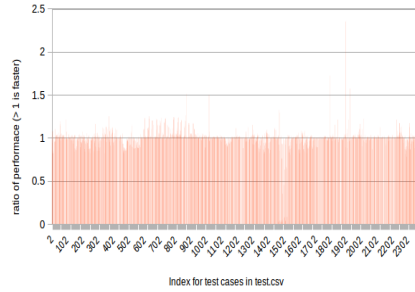
GCM	Total Instructions		Scalars		Loads		Stores	Calls	Instructions
metric	H	R	H	R	H	R	H=R	H	sunk
400.perlbench	2387	2456	855	857	1493	1560	38	1	4
401.bzip2	176	300	27	27	149	273	0	0	0
403.gcc	5670	5761	2558	2567	3109	3191	3	0	1
429.mcf	42	43	5	5	37	38	0	0	0
433.milc	128	130	10	10	117	119	0	1	0
444.namd	262	264	113	113	148	150	0	1	0
445.gobmk	1021	1052	312	314	705	734	3	1	1
447.dealII	10455	12325	4009	4063	6394	8210	20	32	0
450.soplex	2389	2731	927	1004	1437	1702	14	11	0
453.povray	1967	2269	560	590	1389	1661	4	14	0
456.hmmer	408	427	82	82	319	338	1	6	0
458.sjeng	372	399	128	138	234	251	1	9	1
462.libquantum	49	49	17	17	32	32	0	0	0
464.h264ref	2953	3438	712	779	2238	2656	2	1	1
470.lbm	23	23	4	4	19	19	0	0	0
471.omnetpp	391	408	92	92	297	314	0	2	0
473.astar	202	222	62	64	140	158	0	0	0
482.sphinx3	168	172	32	32	136	140	0	0	0
483.xalancbmk	5894	6162	2586	2660	3301	3495	4	3	2
Grand total	34957	38631	13091	13418	21694	25041	90	82	10

Table 5. GCM stats on SPEC2006 at -Ofast, H(Hoisted,) R(Removed); Loads are hoisted the most followed by scalars, stores and calls.

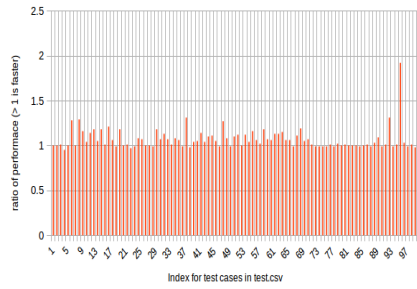
We compiled a popular open source computer vision library, OpenCV. We used OpenCV performance tests and scripts provided by them to measure performance gain with global-code-motion. We compared performance for every test in different test suites. We observe some gains with minor regressions. Feature2D test suite resulted in performance gain of 1 – 2% without any performance regression. Ratio of performance with global-code-motion for different test suites is shown in Figure ??.



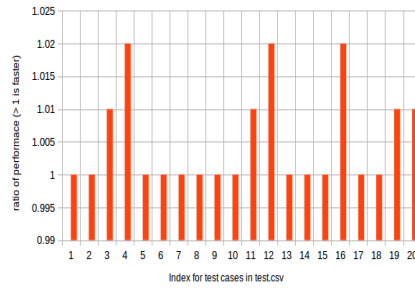
(a) Calib3D



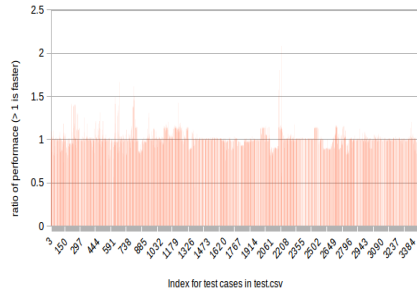
(b) Core



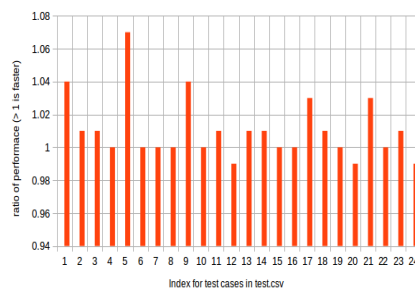
(c) dnn



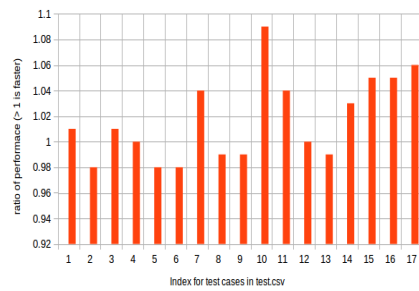
(d) Feature2D



(e) ImgProc



(f) ObjDetect



(g) Video

Fig. 2. Ratio of performance for OpenCV perf test with global-code-motion

We report decrease in size of `libopencv_core` library by 0.17%. There is no regression observed in size of other libraries. We cross-compiled OpenCV for Android using Android ntk 16b. We report decrease in size in Figure ?? . Notably with global-code-motion we observed 0.4% decrease in size for OpenCV machine learning library `libopencv_ml.so*` and it's core library `libopencv_core.so*`.

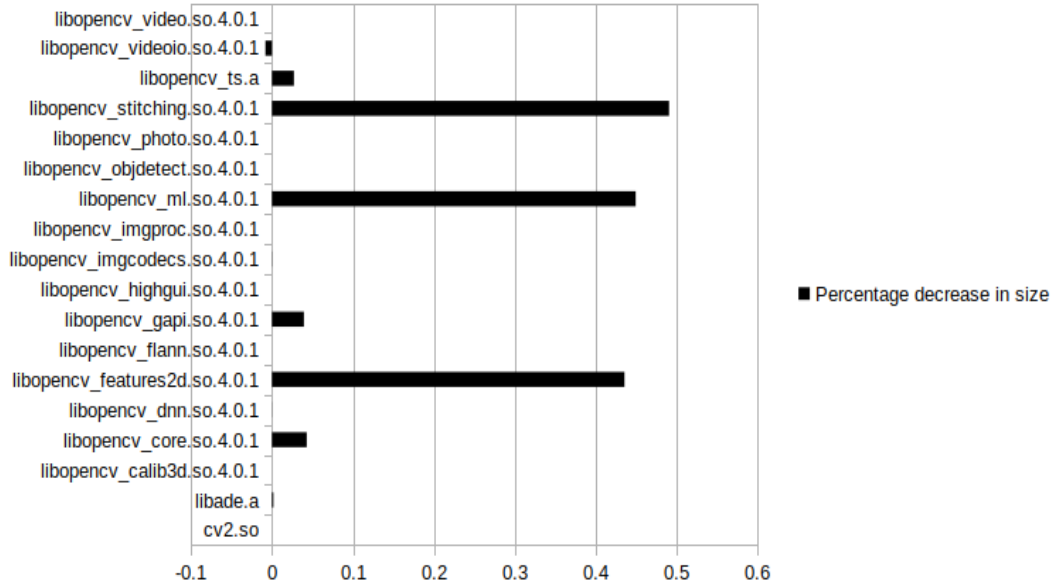


Fig. 3. Percentage decrease in size of OpenCV libraries on Android with global-code-motion

The code shown in Section ?? 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 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 ?. A part of this implementation is already merged in LLVM trunk as `GVNHoist.cpp` and we posted the GCM implementation for review

There are a few improvements that we would like to make. First would be to integrate this 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 is not the most optimal approach in terms of compile time complexity and results in

multiple data structures to be recomputed. This can be improved by ranking the computations (?). 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. The first candidate would be the inliner. We have seen difference in inlining decisions in Table ??, when global-code-motion was enabled. Since inliner has several cost-models tuned for the previous pass layout, that would need some improvement.