GVN-Hoist: Hoisting Computations from Branches

Aditya Kumar
Samsung Austin R&D Center
aditya.k7@samsung.com

Sebastian Pop
Samsung Austin R&D Center
s.pop@samsung.com

ABSTRACT

Code-hoisting identifies identical computations across the program and hoists them to a common dominator so as to save code size. Although the main goal of code-hoisting is not to remove redundancies: it effectively exposes redundancies and enables other passes like LICM to remove more redundancies. The main goal of code-hoisting is to reduce code size with the added benefit of exposing more instruction level parallelism and reduced register pressure.

We present a code hoisting pass that we implemented in LLVM. It is based on Global Value Numbering infrastructure available in LLVM. The experimental results show an average of 2.5% savings in code size, although the code size increases in many cases because it enables more inlining. This is an optimistic algorithm in the sense that we consider all identical computations in a function as potential candidates to be hoisted. We make an extra effort to hoist candidates by partitioning the potential candidates in a way to enable partial hoisting in case common hoisting points for all the candidates cannot be found. We also formalize cases when register pressure will reduce as a result of hoisting.

1. INTRODUCTION

Compiler techniques to remove redundant computations are composed of an analysis phase that detects identical computations in the program and a transformation phase that reduces the number of run-time computations. Classical scalar optimizations like CSE [6] work very well on single basic blocks. When it comes to detect redundancies across basic blocks these techniques fall short: more complex passes like GCSE and PRE have been designed to handle these cases based on dataflow analysis [17]. At first these techniques were described in the classical data-flow analysis framework, and later the use of the SSA representation lowered the cost in terms of compilation time [7, 8, 16] and brought these techniques in the main stream: nowadays SSA based PRE is available in every industrial compiler.

This paper describes code-hoisting, a technique that uses the information computed for PRE to detect identical computations and 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. Code-hoisting is not a redundancy elimination pass, and thus it has different cost function and heuristics than PRE or CSE. Code-hoisting is also different from global code scheduling [6, 9] in the sense that code-hoisting will only hoist computations when there are identical computations sharing a common dominator. The goals of code hoisting are:

- to reduce the code size of the program;
- to improve function inlining heuristics: functions become cheaper to inline by reducing their code size;
- to expose more instruction level parallelism: by hoisting identical computations to be executed earlier, instruction schedulers can move heavy computations earlier in order to avoid pipeline bubbles;
- to help out-of-order processors with speculative execution of branches: by hoisting expressions out of branches, code hoisting can effectively reduce the amount of code to be speculatively executed and can reduce the critical path;
- to reduce register pressure: by moving computations closer to the definitions of their operands;
- to improve passes that do not work well with branches:
  - to improve loop vectorization by reducing a loop with control flow to a loop with a single BB, should all the instructions in a conditional get hoisted and sinked;
  - to enable more loop invariant code motion (LICM): as LICM does not reason about instructions in the context of loops with conditional branches, code-hoisting is needed to move instructions out of conditional expressions and expose them to LICM.

The main contributions of this paper are:

- a new algorithm to hoist computations from branches,
- cost models to reduce live-range and reduce spills,
- performance evaluation of the implementation in LLVM,
- comparison against other algorithms for code hoisting.

LLVM-dev 2016

2. RELATED WORK

There are a lot of bug reports in GCC and LLVM bugzillas [2, 5], showing the interest in having a more powerful code hoist transform. The current LLVM implementation of code hoisting in SimplifyCFG.cpp $HoistThenElseCodeToIf()$ is very limited to hoisting from identical basic blocks: the instructions of two sibling basic blocks are read in the same time, and all the instructions of the blocks are hoisted to the common parent block as long as the compiler is able to prove that the instructions are equivalent. This implementation does not allow for an easy extension: first in terms of compilation time overhead the implementation would become quadratic in number of instructions to bisimulate, and second the equivalence of instructions currently uses

$$I1 -> isIdenticalToWhenDefined(I2)$$

would need to be rewritten to be more general, leading to using a mechanism similar to the idea described in this paper based on GVN.

Rosen et al. explain moving computations successors to remove redundancies [20]. Their algorithm iterates on computations of same rank and move the code with identical computations from the sibling branch. Also there is no notion of partially hoisting the computations so their approach may result in missing many hoisting opportunities.

Dhamdhere [13], Muchnick [18] mention code hoisting in a data flow framework. A list of Very Busy Expressions (VBE) are computed which are hoisted in a basic block where the expression is anticipable (all the operands are available). This algorithm would hoist as far as possible without regarding the impact on register pressure and as such a cost model will be required. Also the description of VBE is based on the classic dataflow model and an adaptation to a sparse SSA representation is required.

GCC recently got code-hoisting [3] which is implemented as part of GVN-PRE: it uses the set of ANTIC and AVAIL value expressions computed for PRE. The algorithm hoists top down in a predecessor where the value is in ANTIC. It uses ANTIC[B] to know what expressions will be computed on every path from B to exit, and can be computed in B: that is the safety condition. It uses AVAIL[B] to subtract out those values already being computed (because they are already available): this is a redundancy condition. The cost function is: for each hoist candidate, if all successors of B are dominated by B, then we know insertion into B will eliminate all the remaining computations. It then checks to see if at least one successor of B has the value available. This avoids hoisting it way up the chain to ANTIC. It also checks to ensure that B has multiple successors, since hoisting in a straight line is pointless. The algorithm continues on down the dominator tree, iterating with PRE until no more changes. One advantage of GCC implementation is that it works in sync with the GVN-PRE such that when new hoisting opportunities are created by GVN-PRE, code-hoisting will hoist them.

3. CODE HOISTING

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

- Control Flow Graph (CFG) and the Dominance (DOM) and Post-Dominance (PDOM) relations [6];
- Single Entry Single Exit (SESE) [15] and Single Entry Multiple Exit (SEME) regions;
- Static Single Assignment (SSA) [12];
- Global Value Numbering (GVN) [20, 9]: to identify similar computations compilers use GVN. Each expression is given a unique number and the expressions that the compiler can prove identical are given the same number;
- Memory SSA [19]: memory operations that the compiler is able to prove in dependence are linked through use-def chains.

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

- find candidates: instructions computing same values,
- compute a point in the program where it is both legal and profitable to move the code,
- transform the code.

3.1 Finding candidates to hoist

The first step is to find a set of instructions that perform identical computations: this is performed by a linear scan of all instructions of the program and classifying all instructions by their value given by GVN.

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 short-time solution to value number loads is to hash the address from where the value is to be loaded. For stores, we value number the address as well as the value to be stored at that address.

Another limitation of the current GVN implementation in LLVM is that the instructions dependent on the loads will not get numbered correctly, and so after hoisting all candidates we need to rerun the GVN analysis in order to discover new candidates now available after having hoisted load instructions. This limitation should be addressed in a new implementation of the GVN based on MemorySSA, that would better account for equivalent loads and their dependent instructions.

The process of computing GVN can be on-demand (as we come across an instruction) or, precomputed (computing GVN of all the instructions beforehand). Which process to choose is determined by the scope of code-hoisting we want to perform. In a pessimistic approach, as described in Section 4.2, we want to hoist a limited set of instructions from the sibling branches as we iterate the DFS tree bottom-up, it is sufficient to compute GVN values on-demand. Whereas, in the optimistic approach, as described in Section 4.1, we want to hoist as many instructions as possible, and it would require GVN values to be precomputed.

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

3.2 Legality check

Since the equality of candidates is purely based on the value they compute, we need to establish if hoisting them to a common dominator would be feasible. Once a common dominator is found, we check whether all the use-operands of the set of instructions are available at that position. In some cases when the operands are not available, it is possible to reinstantiate (rematerialize) the use-operands, thus passing the legality check.

Subsequently, it is checked that the side-effects of the computations does not intersect with any side-effects between the instructions to be hoisted and their hoisting point. For memory operations like loads, stores, calls etc., it is also required to check that all the paths from the hoisting point to the end of the function should execute the exact instruction, in order to guarantee correctness.

Moreover, hoisting of memory operations is tricky on paths which have indirect branch targets e.g., landing pad, case statements, goto labels etc., because it becomes difficult to prove that all the paths from hoisting point to the end of the function would execute the instruction. In our current implementation we discard hoisting through such paths.

In the optimistic approach, described in Section 4.1, it is possible that a common hoisting point of all the instructions is either too far away, or not legally possible. In these cases, it is still possible to 'partially' hoist a subset of instructions by splitting the set of candidates and finding a closer hoisting point for each subset. For more details see Section 4.1.1.

3.2.1 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 computation can be hoisted.

3.2.2 Legality of hoisting loads

The availability of operand to the load (an address) is checked at the hoisting point. If that is not available we try to rematerialize the addresss if possible. Along the path, from current position of the load instruction backwards on the control flow to the hoisting point, we check whether there are writes to memory that may alias with the load, in which case we discard the candidate.

3.2.3 Legality of hoisting stores

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

3.2.4 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 will be handled as described for scalar, load, and store instructions.

3.3 Profitability check

After the legality checks have passed, we check for profitability of hoisting. That takes into account the impact code-hoisting would have on various parameters that affect runtime performance e.g., impact on live-range, gain in the code size. We have established a set of cost models described in Section 5 for each parameter and tuned them for performance against representative benchmarks.

3.3.1 Profitability of hoisting scalars

Since scalars are the majority of instructions which are hoisted, we pay special attention in case of hoisting scalars too far, as that may increase register pressure and result in spills. For example hoisting a scalar past a call, as described in Section 5.2. In our current implementation we hoist scalars past a call only when optimizing for code-side (-Os). Ideally, a later stage of live-range splitting pass should split the live-ranges for optimal performance, however, that is not the case with LLVM as we have found regressions when scalars are hoisted too far, as in Section 5. Another way to mitigate this problem is be to reinstantiate (rematerialize) the computation after a call (may be as a different optimization pass).

3.3.2 Profitability of hoisting loads

A load instruction introduces a register where the value loaded will be kept, the register pressure increases by one (unless the operand to load becomes dead at the load). On the other hand, loading a value early will reduce the stall during execution should the value is not in the cache. We generally prefer to hoist load except the hoisting point is too far (this distance is computed by looking at the experimental results of representative benchmarks, see Section 6).

3.3.3 Profitability of hoisting stores

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

3.3.4 Profitability of hoisting calls

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

3.4 Code generation

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

After one iteration of algorithm runs through the entire function, it creates more oppportunities for *higher ranked* computations [20]. 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 [20], or by improving the GVN analysis to correctly number loads and dependent instructions.

Finally after the transform 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.

3.5 Illustrative Example

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

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

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

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

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

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

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

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

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

  return tmax + tmin;
}
```

In this code the two subtractions and the division operations can be executed in parallel because there are no dependencies among them. So the total number of cycles will be $max(C_{div}, C_{sub}) + C_{mul} = C_{div} + C_{mul}$; since C_{div} is usually much greater than C_{sub} [4, 1].

4. CODE HOISTING POLICIES

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, however, that analysis is beyond the scope of this paper.

4.1 Hoisting bottom up with optimistic approach

In this approach, the goal is to maximize the total number of hoistings in the entire function. This algorithm is very useful when optimizing for code-size. We collect the GVN of all the instructions in the function and iterate on the list of instructions having identical GVNs. After that we find the common dominator dominating all such identical computations and perform legality checks, as described in Section 3.2. 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., hoisting point too far. In those cases, this algorithmn would partition the list of identical instructions into subsets which can be partially hoisted to their respective common dominators. The partition algorithm is described as follows:

4.1.1 Partition the list of hoisting candidates to maximize hoisting

In order to hoist a subset of identical instructions, we partition the list of all candidates in a way to maximize the total number of hoistings. By sorting the list of all the candidates in the increasing order of their depth first search discovery time stamp [11] (DFSIn numbers), we make sure that candidates closer in the list have their common dominator nearby in cases when there are no fully redundant instructions. Essentially if,

```
// B1 != B2 != B3
BasicBlock B1, B2, B3
DFSIn(B1) = depth first discovery time stamp
DFSIn(B1,B2) = |DFSIn(B1) - DFSIn(B2)|
DFSIn(B1,B3) = |DFSIn(B1) - DFSIn(B3)|

Depth(B1) = depth of tree from the root node.
// When B1 dominates B2
BBDist(B1, B2) = |Depth(B1) - Depth(B2)|

NCD(B1, B2) = nearest common dominator of B1 and B2
BBDist(B1,B2) = BBDist(B1, NCD(B1, B2))
BBDist(B1,B3) = BBDist(B1, NCD(B1, B3))

Then,
DFSIn(B1,B2) < DFSIn(B1,B3) implies
 BBDist(B1,B2) <= BBDist(B1,B3)
```

Same property would hold for instructions I1, I2, I3 provided they are not fully redundant, i.e., none of the basic blocks containing I1, I2 and I3 respectively dominate the other:

```
Instruction I1, I2, I3
// I1, I2, I3 are not fully redundant
// B1, B2, B3 contains I1, I2, I3 respectively
DFSIn(I1) = DFSIn(B1)
BBDist(I1, I2) = BBDist(B1, B2)
DFSIn(I1) = DFSIn(B1)
DFSIn(I1,I3) = DFSIn(B1, B3)
NCD(I1, I2) = NCD(B1, B2)
```

```
Then,
 DFSIn(I1,I2) < DFSIn(I1,I3) implies
  BBDist(I1,I2) <= BBDist(I1,I3)
```

It says that if I1 is closer to I2 than I3 in the list of candidates sorted by DFSIn numbers, then the nearest common dominator of I1 and I2 will be closer (in terms of basic block distance) to I1, than nearest common dominator of I1 and I3.

In the presence of fully redundant computations in the list of hoistable candidates this equation may not hold. For example, if there is another instruction I4 which is dominated by I1, but I4 is present in a basic block which is farther than $DFSIn(I1, I2)$, i.e., $DFSIn(I1, I2) < DFSIn(I1, I4)$; however, $BBDist(I1, I4) = 0 < BBDist(I1, I2) >= 1$.

So by sorting candidates w.r.t. their DFSIn numbers:

1. would make fewer checks for legality and profitability.
2. hoisting point will be closer, so the intersection of live-range of the instruction with other instructions will be minimal.

In our current implementation we keep as many candidates in one set as possible (greedy approach). We split the list at a point where the legality checks fail to hoist subset of candidates which are legal to hoist and then start finding new hoisting point for the remaining ones. So there are two limitations of the current implementation of the partition algorithm: first, sorting by DFSIn numbers does not give the desired order when there are fully redundant instructions; second, it lacks cost model for partitioning in order to preserve/improve performance, see Section 7.

4.2 Hoisting bottom up with pessimistic approach

In the pessimistic approach, the basic blocks are traversed in the inverse depth-first order, computing the GVN of instructions as they come by. The GVN of sibling branches are compared for equality. Once such a candidate is found, it is hoisted in the common dominator. All the leaf nodes are visited before the non-leaf nodes (bottom-up) because instructions are hoisted upwards.

The pessimistic algorithm is fast, results in fewer spills and also hoists fewer instructions. We have implemented optimistic approach because it is more general and can be tuned down to closely mimic pessimistic approach by changing a flag.

4.3 Time complexity of algorithm

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

5. COST MODELS

Similar to any compiler optimization pass, there are several cost functions that are deployed to tune for optimal combination of performance and code-size. Since this is mostly a code-size optimization pass, the goal is to not regress in performance across popular benchmarks at the same time reduce code size as much as possible. Following are the cost models which are implemented:

5.1 Reduce register pressure

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

```
B0: P0: b = 1
    goto B1

B1: P1: c = 2
    goto B2

B2: P2: if c is true then goto B3 else goto B4

B3: P3: a0 = b + c
    goto B5

B4: P4: a1 = b + c
    goto B5

B5: P5: d = phi {a0(B3), a1(B4)}

If we measure D(Px,Py) as total instruction
count in the path from the position of Px to Py

live-range(a0) = D(P0,P3) + D(P1,P3) + D(P3,P5)
               = 6 + 4 + 2 = 12
live-range(a1) = D(P0,P4) + D(P1,P4) + D(P4,P5)
               = 6 + 4 + 2 = 12

old-live-range = max(live-range(a0),live-range(a1))
               = 12

After hoisting a0 and a1 are removed and a copy
of a0 as a01 is placed in B2 just before P2.

live-range(a01) = D(P0,P2-1) + D(P1,P2-1) + D(P2 -1,P5)
                = 4 + 3 + 3 = 10
new-live-range = live-range(a01)
```

If the new live-range is less than the old one it will be a good candidate for hoisting. The live ranges can remain same as well if there is only one operand on the right hand side, e.g., an assignment operation or, one of the operands is not a register. That means hoisting upwards will decrease the live-range of its use but increase the live-range of its definition.

In a special case where the instruction to be hoisted has the last use of its operands then the code hoisting will always reduce the register pressure if it has two register operands because the gain in live-range will be in the ratio of 2:1. Based on the above formulae we can also deduce that, as long as there is one register operand in the right hand side with its last use, code hoisting will either decrease or preserve the register pressure.

5.2 In the presence of calls

Hoisting scalars across calls is tricky because it can increase the number of spills. During the frame lowering of

calls, the argument registers, in general, the caller saved registers are saved because they might be modified by the callee and after the call they are restored. So before the call, the register pressure is high because the number of available registers are reduced by the number of caller saved registers. In that situation if a computation is hoisted across the call, that would increase the total number of registers required by one, thus contributing to the register pressure. However, in the special case discussed in Section 5.1, it will be okay to hoist because the register pressure would not decrease.

Hoisting loads/stores across calls also require precise analysis of all the memory addresses accessed by the call. Our implementation being an intraprocedural pass, the analysis is very conservative. In the presence of pure calls, loads can be hoisted but stores can't. Also, if the call throws exceptions, or if it may not return, memory references cannot be hoisted.

5.3 Hoisting too far away

If there are several instructions in between the hoisting point and the instruction to be hoisted, the instruction to be hoisted crosses several instructions while hoisting, that means we are adding one register to all the live-ranges spanning the instructions. That could result in spills. In the current implementation we choose to hoist if the number of instructions crossed is below a threshold. Ideally, it should be okay to hoist all the instructions and a later a live-range-splitting [10] pass should make the right decision of rematerializing the instruction should it be beneficial to do so. But the current live-range splitting pass of LLVM is not making the optimial decision and we have found spills if the threshold is exceeded. The threshold was computed as a result of tuning the LLVM test-suite, SPEC Cpu2000, and Cpu2006 benchmarks [14].

Also, hoisting a load increases the register pressure by one across all the instructions which the load would cross. That could result in spills later in the register allocation.

However, in the special case discussed in Section 5.1, it will be okay to hoist because the register pressure would not decrease.

6. EXPERIMENTAL EVALUATION

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

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

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

While benchmarking LLVM test-suite we see both increase as well as decrease in the codesizes of the final binaries. Since the pass runs early, it affects many optimizations which rely on the number of instructions, lenth of the use-def chain, and other metrics. For instance, the inliner is impacted by a decrease in the number of instructions in the caller and callee, as its heuristics estimate the size of functions to be inlined. Various code-size metrics are shown in Table 3. All but one benchmark varied between -5.32% and 5.43%. In one benchmark FreeBench/distray/distray.test, the codesize increased by 35.38%. In this benchmark 3 more functions got inlined (15 as compared to 12) and because of that 10 more vector instructions got generated (81 vs. 71), 3 calls got hoisted/sunk as (compared to 0), one loop got unswitched (compared to 0), 6 high latency machine instructions got hoisted out of loop, 59 (compared to 30) machine instructions got hoisted out of loop, 70 (compared to 39) machine instructions were sunk.

The code shown in Section 3.5 is a reduced example that appears in a hot loop of a proprietary benchmark. When the expressions are hoisted from the conditional clauses, the

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

Table 1: Code hoisting metrics on LLVM test-suite

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

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

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

Table 3: Code size metrics on LLVM test-suite

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.

7. CONCLUSION AND FUTURE WORK

We have presented the GVN based code hoisting algorithm. The primary goal is to reduce the code size but it benefits performance in some cases as well. To preserve performance and not hoist too much we have implemented several cost models described in Section 5. Since those cost models depend on a set of thresholds, it requires tuning, as such, we used representative benchmarks to tune them.

Currently, we rerun the algorithm until there are no more instructions left to be hoisted. This is not the most optimal approach and results in expensive analyses to be recomputed. This can be improved by ranking the computations [20]. Also GVN-hoist runs very early in the pass pipeline, it will be good to evaluate the codesize/performance impact when it is run in sync with GVN-PRE just like GCC does.

With the implementation of code-hoisting in LLVM, the passes which rely on the code-size/instruction-count to make optimization decisions needs to be revisited. The first candidate would be the inliner. We have seen different inlining decisions in Table 3, before and after code-hoisting was enabled. Since inliner has several magic numbers tuned for the previous pass layout, it would need some improvement.

8. ACKNOWLEDGMENTS

We would like to thank Daniel Berlin for his code reviews and for his feedback on earlier versions of this paper.

9. REFERENCES

[1] Arm cortex-a57 software optimization guide. http://infocenter.arm.com/help/topic/com.arm.doc.uan0015b/Cortex_A57_Software_Optimization_Guide_external.pdf.

[2] GCC bugs related to code hoisting. https://gcc.gnu.org/PR5738 , https://gcc.gnu.org/PR11820, https://gcc.gnu.org/PR11832, https://gcc.gnu.org/PR21485, https://gcc.gnu.org/PR23286, https://gcc.gnu.org/PR29144, https://gcc.gnu.org/PR32590, https://gcc.gnu.org/PR33315, https://gcc.gnu.org/PR35303, https://gcc.gnu.org/PR38204, https://gcc.gnu.org/PR43159, https://gcc.gnu.org/PR52256.

[3] GCC code hoisting. https://gcc.gnu.org/viewcvs/gcc?view=revision&revision=238242.

[4] Intel 64 and ia-32 architectures optimization reference manual. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[5] LLVM bugs related to code hoisting. https://llvm.org/bugs/show_bug.cgi?id=12754, https://llvm.org/bugs/show_bug.cgi?id=20242, https://llvm.org/bugs/show_bug.cgi?id=22005.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*. Addison wesley, 1986.

[7] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. In *ACM SIGPLAN Notices*, volume 29, pages 159–170. ACM, 1994.

[8] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on ssa form. In *ACM SIGPLAN Notices*, volume 32, pages 273–286. ACM, 1997.

[9] C. Click. Global code motion/global value numbering. In *ACM SIGPLAN Notices*, volume 30, pages 246–257. ACM, 1995.

[10] K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocator. In *International Conference on Compiler Construction*, pages 174–187. Springer, 1998.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[13] D. M. Dhamdhere. A fast algorithm for code movement optimisation. *ACM SIGPLAN Notices*, 23(10):172–180, 1988.

[14] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, July 2000.

[15] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *ACM SigPlan Notices*, volume 29, pages 171–185. ACM, 1994.

[16] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in ssa form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):627–676, 1999.

[17] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[18] S. S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[19] D. Novillo. Memory ssa-a unified approach for sparsely representing memory operations. In *Proc of the GCC Developers' Summit*. Citeseer, 2007.

[20] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM, 1988.