Implementing Profile-Guided Speculative Code Motion in LLVM [*]

Ben Jaiyen
bjaiyen@cmu.edu

Jamie Liu
jamiel@cmu.edu

02 May 2012

1. Introduction

Partial redundancy elimination [10] (PRE) is a class of compiler optimizations that identifies and removes redundant expressions on some execution paths. It does this by searching for expressions that are present on some, but not all, program paths and inserting copies of the expression on the paths that do not compute it such that it becomes fully redundant.

To ensure that optimized code is not slower than the original code, most PRE algorithms will only copy an expression in a program to a new location if it is known that the value calculated by that expression will be used on all paths leading out of its new location. In other words, the expression must be *fully anticipated*, or *downsafe*, on all paths so that the compiler knows it will not lengthen some critical path of the program when moving code. Also, there is no reason to move code to some location if the expression is already *available* at that location. That is, if some previous calculation of an expression can still be used at a point considered for code motion, there is no reason to repeat the calculation again.

Figure 1a shows an example CFG in which an expression *a+b* is partially redundant at the block labeled *Uses a+b*. One of the paths leading up to the block *Uses a+b* calculates *a+b*, i.e. *a+b* is *partially available* at *Uses a+b*. Typical PRE would try to copy the expression *a+b* into the other path, but this would violate anticipation as a calculation of *a+b* is being introduced into a path that it was not calculated in previously as Figure 1b illustrates. Since *a+b* is only *partially anticipated* as opposed to being *fully anticipated*, the compiler cannot safely move the expression.

By utilizing profile information, we can ignore this notion of anticipation by relying on a probabilistic argument. If a real run of the program tells us that an expression will be used many times on some paths leading out of a basic block, and not used a few times on other paths leading out of the same basic block, then ignoring anticipation at this basic block could lead to performance gains as the critical path is only lengthened a few times whereas the amount of computation done is reduced most of the time. Figure 1c shows there is a good chance that breaking anticipation will lead to performance speedups in our example as we are able to eliminate a redundant expression that will lead to more performance gains than what we lose by increasing the rarely executed critical path by a little. This type of optimization is known as speculative code motion as there is still a chance that it will degrade performance if, for example, the profile run of the program being optimized was not representative of all input sets that the program might receive.

Our mechanism modifies an existing PRE algorithm, GVN-PRE [11], to use profile information as a metric for guiding speculative code motion. It violates the downsafety requirement in cases where the profile information indicates that we can reduce the amount of overall computation despite introducing some useless computations.

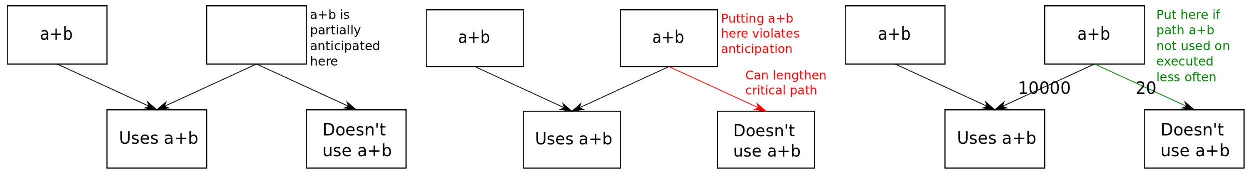
1.1. Related Work

Lazy code motion [8], proposed by Knoop et al, is a PRE algorithm that will not move an expression unless it is fully anticipated at its new location. This guarantees that the optimized code will not run slower than the original code, or in other words, it produces computationally optimal results.

Cai and Xue use edge profile information along with the minimum cut algorithm to minimize the amount of expression computations in PRE [4]. Their mechanism, MC-PRE, removes edges and nodes from the flow graph that do not affect the PRE computations to create a reduced flow graph. MC-PRE then calculates the minimal cut on the reduced graph to find the optimal insertion points.

Kennedy et al. describe an algorithm, SSAPRE [7], that adapts PRE to programs that are represented in single static assignment (SSA) form [5]. Zhou et al.'s algorithm, MC-SSAPRE, expands on SSAPRE by converting it into a flow network problem similar to what is done in the MC-PRE algorithm. Contrary to MC-PRE, MC-SSAPRE operates on expressions one at a time, which allows it to form smaller flow graphs

[*]http://www.ece.cmu.edu/~jamiel/15-745/


a+b
a+b is partially anticipated here
Uses a+b
Doesn't use a+b
a+b
a+b
Putting a+b here violates anticipation
Can lengthen critical path
Uses a+b
Doesn't use a+b
a+b
a+b
Put here if path a+b not used on executed less often
10000
20
Uses a+b
Doesn't use a+b

(a) Example CFG

(b) Violating anticipation with normal PRE

(c) Using Profile Information (Edge weights are execution counts)

Figure 1: Example of how profile information can be used to violate anticipation smartly

and more efficiently compute the minimum cut.

VanDrunen et al. proposed global value numbering, an SSA based optimization that eliminates redundant code [11]. It determines values produced by expressions and assigned to variables, and removes redundancy by looking at equal values.

1.2. Contributions

Our contributions are as follows:

- We augment the GVN-PRE algorithm by utilizing profile information to perform speculative code motion.
- We describe a method of creating a flow network graph from information gathered during GVN-PRE passes.
- We implement our mechanism in LLVM Compiler Infrastructure [9] and evaluate the effectiveness of it.

2. Design and Approach

Despite being developed specifically for programs in SSA representation, SSAPRE only attempts to eliminate redundancies between *lexically identified* expressions (that is, expressions that share operand variable names in the original, non-SSA source code). Aside from limiting optimization potential, this is especially problematic in the context of LLVM, which does not retain information about source variables in its SSA representation. Previous work [3] attempted to reverse-engineer the LLVM SSA representation using a heuristic to find source variables, but their implementation was outperformed by LLVM's existing GCSE and LICM optimizations, suggesting that this heuristic was ineffective.

SSAPRE places two constraints on the SSA representation. First, “each ϕ assignment [must have] the property that its left-hand side and all of its operands are version of the same program variable” [7]. Second, “the live ranges of different version of the same original program variable [must] not overlap” [7]. These restrictions severely restrict what optimizations may be performed prior to applying SSAPRE. In particular, [3] observed that the LLVM `mem2reg` optimization, which promotes (among other things) stack variable accesses to SSA registers, causes these constraints to be violated. However, without the `mem2reg` optimization, it is very difficult to do any optimization since most interesting optimization opportunities are hidden behind loads and stores to stack variables. We conjecture that this is part of why [3] was unable to provide optimization benefits on par with GCSE / LICM.

Our mechanism is focused on adapting the key idea in MC-SSAPRE to the GVN-PRE framework [11]. GVN-PRE (global value numbering / partial redundancy elimination) is a newer technique for performing PRE on programs in SSA form that is both more powerful than SSAPRE (working on redundant values rather than lexically identified expressions) and easier to understand. GVN-PRE also represents the state of the art; the most current version of GCC, GCC 4.7, implements it [1],[1] and LLVM implemented it up to version 2.6, after which it was dropped due to lack of maintenance (and not replaced) [2].

Our mechanism, which we call MC-GVN-PRE, builds on top of the GVN-PRE mechanism in the following ways. We first create a reduced flow graph for each PRE candidate expression by using the GVN-PRE analysis to determine when an expression is partially/fully anticipated and when it is used. GVN-PRE only moves expressions to points where they are fully anticipated. We allow the movement of expressions to points where they are partially anticipated provided they do not have side effects. The reduced flow graph is then augmented with artificial source and sink nodes. The artificial source node is placed to

[1]Notably, GCC's implementation of GVN-PRE replaced an earlier implementation of SSAPRE.

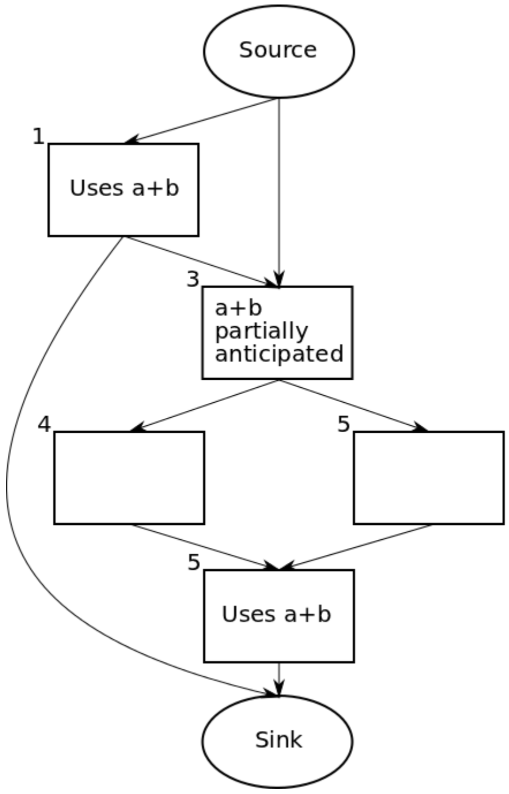


Figure 2: Flow graph construction example

prevent an expression from being introduced above where it is last fully unavailable. (This is consistent with both GVN-PRE and MC-SSAPRE, and leads to lifetime optimality, i.e. the live ranges of any computations introduced is minimized. Placing the expression at an earlier point may lead to reduced code size, but we leave this for future work.) The artificial sink node is placed after each use of the expression, ensuring that placing the expression at the minimum cut will result in the expression being available at each use. By calculating the minimum cut of this reduced flow graph, we can determine the optimal points to place the expression. To find the minimum cut of our flow graph, we use the Ford-Fulkerson algorithm [6]. Similar to MC-SSAPRE, we also choose the minimum cut closest to the sink to minimize the lifetimes of temporaries.

Figure 2 shows an example of a flow graph that might be constructed by our mechanism. Our initial flow graph consists of the entire program CFG. It is pruned to remove nodes that the expression cannot be placed in and an artificial source and sink are introduced to complete the flow graph. The source is attached to blocks 1 and 3 as the expression is not available above them. Blocks 1 and 5 contain uses of the expression and therefore are hooked up to the artificial sink. The edge weights in the flow graph are determined by profiling and recording execution counts. Once the minimum cut has been determined, copies and ϕ nodes are placed above and below the cut as per ordinary GVN-PRE.

In summary, MC-GVN-PRE compares to GVN-PRE as follows:

- The first step in GVN-PRE, `BuildSets`, computes availability and full anticipability for each basic block. We augment this step to additionally compute partial availability and partial anticipability (whereas full availability and anticipability provide guarantees on execution-independent optimality, partial availability and anticipability are necessary to establish optimality under an execution profile, as discussed above.)
- The second step in GVN-PRE, `Insertion`, examines each merge point in the function and performs partial redundancy elimination by inserting computations and ϕ nodes wherever necessary. MC-GVN-PRE works much differently, since it must work on one expression at a time. As such, `Insertion` in MC-GVN-PRE proceeds as follows: For each expression,

  - Construct a directed *reduced flow graph* that is initially equivalent to the control flow graph with edges weighted by branch frequency (as determined by profiling).
  - An artificial sink node is added to the graph. This node serves two purposes. First, as in MC-SSAPRE, it maintains the invariant that computations precede their uses. This is done by creating an edge from nodes containing uses of the expression to the sink node with infinite edge weight (such that the minimum cut will never cross it). Second, it ensures the minimum cut algorithm does not waste time investigating regions of the control flow graph in which the expression under examination is not partially anticipated (i.e. useless). This is done by walking the dominance tree, finding nodes at which the expression is not partially anticipated, and replacing all of the nodes strictly dominated by these nodes with an infinite-weight edge to the sink. (The initial node must be retained, as its edge weight may affect the final minimum cut.)
  - An artificial source node is added to the graph. This node ensures lifetime optimality (that is, added computations never have a longer live range than necessary). This is done by connecting all nodes at the frontier of partial availability (i.e. where the expression has previously been computed at least once) to the source node.
  - The minimum cut in the graph is computed via the Ford-Fulkerson maximum-flow algorithm and the min-cut/max-flow theorem.
  - Insertion is done at nodes at the frontier of the cut using the same algorithm as in GVN-PRE.
- The final step in GVN-PRE, `Elimination`, eliminates fully-redundant expressions, including those introduced by `Insertion`. We do not modify this step.

3. Methodology

The system configuration that we used in our evaluations is given in Table 1. We chose to evaluate our mechanism with microbenchmarks as they are much simpler to analyze and gain insight from. We compile our microbenchmarks using Clang with the -O0 flag and run the LLVM `mem2reg` pass on them before applying our MC-GVN-PRE pass.

Table 2 describes the two characteristics of our microbenchmarks that we believe are the most important to determining a program's potential benefit with speculative code motion. If a program is able to remove more redundancies with a speculative code move, or if moving an expression has a minimal effect on the paths in which the expression is not anticipated, then the execution time will most likely be minimized.

The structure of each microbenchmark consists of an outer loop that runs for a billion iterations and a unique control flow graph inside the loop which tests a different optimization aspect of MC-GVN-PRE.

4. Evaluation

We implemented MC-GVN-PRE and ran the LLVM pass on some microbenchmarks, but some of the microbenchmarks caused it to fail and we were not able to track down the cause of the failure in time. For the microbenchmarks that did succeed , we were able to verify that our mechanism was able to perform PRE correctly and optimally. To get an idea of what kind of results to expect after running a successful LLVM pass we hand optimized the remaining benchmarks with the MC-GVN-PRE mechanism.

Figure 3 shows the performance difference between MC-GVNPRE and a baseline that does not eliminate redundancy after we ran each microbenchmark, using the unix time utility to measure performance. Performance improves by at most 5% and degrades by at most 7% on our set of microbenchmarks. The reason that it degrades so much in benchmark 6 is because the moved expression ends up increasing the critical path of the program 25% of the time. This is not enough to offset the gains from reducing redundancy. This microbenchmark was written in order to show the possible adverse affects of profile guided code motion and the results support our hypothesis. Benchmarks 1 and 2 do well for different reasons. Five redundancies are removed every loop cycle for benchmark 1 whereas benchmark 2 doesn't lose as much performance from lengthening its critical path.

We were able to confirm our earlier insights with regards to MC-GVN-PRE. Trading off reduced redundancy with increased critical path length works up to the point where the critical path begins to negate the beneficial effects of speculative code motion. However, if profiling information is accurate, and some branches have highly skewed execution counts, it is potentially worth taking a small gamble for better

Component	Specifications
Processor	2.66 GHz, 32KB per-core L1, 8MB shared L2
Main Memory	4GB DDR2 667 MHz
Operating System	Ubuntu 11.10

Table 1: Evaluated system configuration

Microbenchmark Number	Redundancies Per Loop	% of time Critical Path Lengthened
1	5	10%
2	1	10%
3	3	5%
4	3	3.33%
5	3	1%
6	3	25%

Table 2: Microbenchmark Characteristics

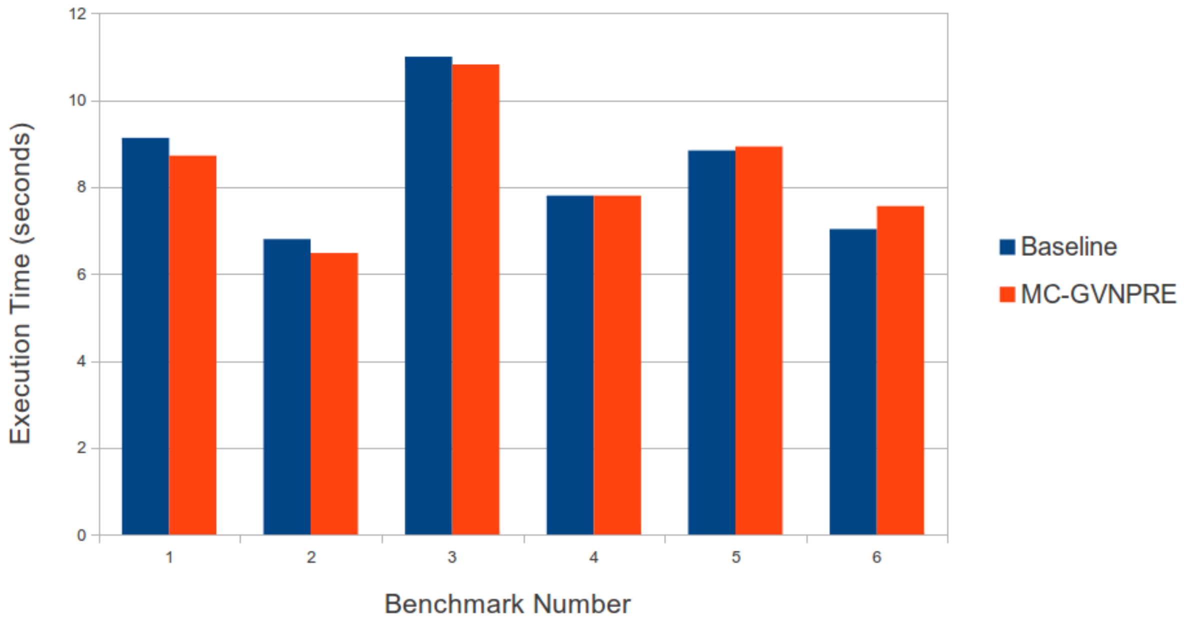


Figure 3: Microbenchmark Performance Results

overall performance.

5. Lessons Learned

The most unexpected result was observed in microbenchmark 5 which increases the critical path ten percent of the time while removing one redundancy every loop iteration. This benchmark consists of an if statement embedded within a for loop that executes for many iterations. An expression is calculated on one branch of the if statement and later recalculated after the if statement merges back into the rest of the control flow. Performance ended up degrading after making this expression redundant on both paths of the if statement. Although PRE optimizes for number of computations, it does not reflect other practical issues such as the effect of code density on instruction and micro-operation caches, as well as the effect of variable lifetimes on register pressure.

6. Conclusion

It has been shown in both our work and the work of others that speculative code motion has the potential to improve performance in some cases. As with most profile guided optimizations, MC-GVN-PRE is heavily reliant on having an accurate profile as differing input sets can cause program behavior to change drastically in some cases. As future work, we would like to further explore other heuristics that can be used to determine when it is valuable to speculatively move code and when it is not.

References

[1] "GVN-PRE." [Online]. Available: http://gcc.gnu.org/wiki/GVN-PRE

[2] "GVNPRE removed from main line?" [Online]. Available: http://old.nabble.com/GVNPRE-removed-from-main-line--td26383411.html

[3] T. Brethour, J. Stanley, and B. Wendling, "An LLVM implementation of SSAPRE," 2002.

[4] Q. Cai and J. Xue, "Optimal and efficient speculation-based partial redundancy elimination," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO '03, 2003, pp. 91–102.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[6] D. R. Ford and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ, USA: Princeton University Press, 2010.

[7] 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*, vol. 21, no. 3, 1999.

[8] J. Knoop, O. Rüthing, and B. Steffen, "Lazy code motion," in *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, ser. PLDI '92, 1992, pp. 224–234.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, 2004.

[10] E. Morel and C. Renvoise, "Global optimization by suppression of partial redundancies," *Commun. ACM*, vol. 22, no. 2, pp. 96–103, Feb. 1979.

[11] T. VanDrunen and A. L. Hosking, "Value-based partial redundancy elimination," in *Proceedings of the 13th Internation Conference on Compiler Construction*, 2004.