An Implementation of GVN-PRE in LLVM
(Project Proposal for CS7968)

Prashanth Radhakrishnan
shanth@cs.utah.edu

Abstract

The Low-Level Virtual Machine [4] (LLVM) compiler infrastructure currently lacks the partial redundancy elimination [5] (PRE) transformation. This is a proposal to enhance LLVM with the GVN-PRE [7] algorithm. GVN-PRE is a technique that uses global value numbering [6] (GVN) to eliminate partially redundant computations. The unified hash-based GVN scheme [1], a by-product of implementing GVN-PRE, by itself may also be useful to existing LLVM transformations.

1 Introduction

The Low-Level Virtual Machine (LLVM) is an infrastructure that provides a collection of libraries and tools for building compilers, optimizers, just-in-time code generators and other compiler-related programs. LLVM defines a language-independent virtual instruction set in Static Single Assignment[1] (SSA) form, both as an offline code representation (to communicate code between compiler phases and to run-time systems) and as the compiler internal representation (to analyze and transform programs). LLVM leverages this representation to provide high-level information to compiler transformations at compile-time, link-time, run-time and idle-time (the latter two are not as popular as the former two).

LLVM currently supports over 100 program analysis and transformation passes. Surprisingly, it does not include the PRE transformation. An implementation of PRE based on the SSAPRE algorithm was attempted in the past [2], but it was incomplete and eventually ignored. GVN-PRE is perceived to be a simpler algorithm and has been implemented in GCC.

[1]An intermediate representation in which every variable is assigned exactly once

2 PRE

Partial redundancy elimination (PRE) removes computations that are redundant on at least one execution path. PRE is considered to be a powerful transformation because it subsumes a (full redundancy elimination) transformation like common subexpression elimination (CSE) and loop invariant code motion (LICM) [5].

In the example *eg1* below, the computation of the expression $(a + b)$ is partially redundant because it is redundant on the path $1 \rightarrow 2 \rightarrow 5$, but not on the path $1 \rightarrow 4 \rightarrow 5$. PRE works by first introducing operations that make the partially redundant expressions fully redundant and then deleting the redundant computations. The computation of $(a + b)$ is added to 4 and then deleted from 5.

```
eg1:
    (1)    if (OPAQUE)
    (2)        x = a + b;
    (3)    else
    (4)        x = 0;
    (5)    y = a + b;
```

Expressions are deemed redundant based on their lexical (or syntactic) equivalence or static values. In the example above, both techniques identify the redundant $(a + b)$. Whereas, in *eg2* below, only static values can help identify the partially redundant computations at 2 and 6, because they are not lexically alike. Global value numbering (GVN) [6] is the technique that assigns unique value numbers to variables and expressions (across basic blocks) having the same static value.

```
eg2:
    (1)    if (OPAQUE)
    (2)        x = a + b;
    (3)    else
    (4)        x = 0;
    (5)    c = a;
    (6)    y = c + b;
```

While early PRE techniques used syntactic equivalence for redundancy detection, the recent ones use value numbering (with better results) [8]. And GVN-PRE [7], as its name suggests, is of the newer breed.

3 GVN-PRE

GVN-PRE [7] works on the Static Single Assignment (SSA) representation of the program. It assumes the presence of a dominator (and post-dominator) graph modelling the *dominance*[2] (and *post-dominance*[3]) relation and the absence of *critical edges*[4] in the control flow graph. Since GVN-PRE uses value numbering, it ignores source-level lexical constraints.

GVN-PRE partitions all expressions in the program into values. These are stored in a value table that maps expressions to value numbers [1] based on their structure (e.g., $value[a+b] = value[a]+value[b]$).

The GVN-PRE algorithm works in three steps.

- In the first step, it runs a system of data-flow equations (over the dominator and post-dominator graphs) to calculate the *available* and *anticipated* expression values at the end and beginning of each basic block respectively. The available values at a program point give all the values computed in the past and this helps PRE identify redundant computations. The anticipated values at a program point give the values that will definitely be computed in the future and this helps PRE identify safe insertion points that will convert partial redundancy to full redundancy. Due to the presence of back-edges, these values are calculated until fixed-point.

- In second step, partially redundant expressions are hoisted to earlier program points to achieve full redundancy. The algorithm considers all merge points with more than one predecessor (i.e., program points where partial redundancy is observed). For each anticipated expression at the merge point, if that expression is available at atleast one of the predecessors, then the expression is hoisted to the predecessors where it is not available. Each such hoisting is followed by traversals of the dominator graph to bubble the changes (until fixed-point).

- In the final step, the algorithm iterates through the program to check if a redundant calculation can be replaced by an assignment. For each instruction, it calculates the expression's value. It checks whether the target in the instruction corresponds to the first occurance (or leader) of this value in the program. If not, the computation is replaced with an assignment of the first occurance variable (or leader of that value) to the target.

To keep the presentation short, the description above left out some important details [5] of the GVN-PRE algorithm [7].

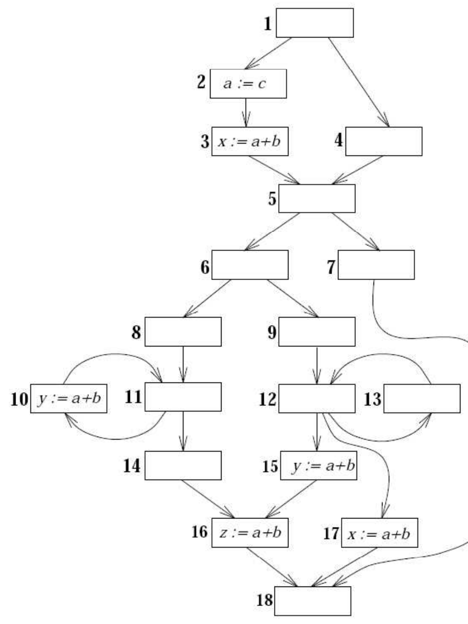

Figure 1: A non-trivial partial redundancy example.

4 Implementation Outline

LLVM does not include a PRE pass. Simple partial redundancies such as *eg1* are eliminated[6] by LLVM.

[2]A node d dominates a node n if every path from the start node to n must go through d.

[3]A node d post-dominates a node n if every path from n to the stop node must go through d

[4]Edges from blocks with more than one successor to blocks with more than one predecessor

[5]a) Phi translation while computing the anticipated sets. b) Tracking leaders and anti-leaders, which are the chosen variables (or expressions) representating values in available and anticipated sets respectively. c) Tracking dependency between the anticipated values. This helps in handling second order effects during code-hoisting and removal of values that get "killed" (by assignment of an "opaque" value).

[6]In *eg1*, LLVM eliminated the partial redundancy by duplicating all the code (not just the $(a + b)$ instruction) after the *if*

Whereas, a non-trivial partial redundancy, like figure 1 (taken from the Lazy Code Motion paper [3]), is not eliminated.

LLVM satisifies the pre-requisites of the GVN-PRE algorithm, namely, SSA-based code representation and presence of passes such as dominator construction, post-dominator construction and critical edges removal. The LLVM pass infrastructure provides passes with access to the program in different ways (iterating over basic-blocks, functions, call-graph order). Since the scope of GVN-PRE is intra-procedural, processing the program one function at a time seems natural. Further, each LLVM pass can declare the set of passes that are required to be executed before the current pass, and the passes which are invalidated by the current pass. We would specify the three passes required by GVN-PRE. Since GVN-PRE does not add or delete basic-blocks, it should preserve the CFG and other structure construction analyses. I have not yet investigated the other transformations that can get invalidated.

Here is the break-up of implementation.

- Implement the hash-based GVN analysis, the first step of the GVN-PRE algorithm. It seems that this GVN analysis may be useful by itself. It may be possible for other transformations to use the results of the GVN analysis. This could be an improvement to LLVM's current value numbering implementation[7]. To demonstrate this improvement, the results of the global common subexpression evaluation (GCSE) pass using both the value numbering schemes could be compared.

- Once GVN analysis works as expected, add code-hoisting and computation elimination to complete the GVN-PRE implementation. At this step, running GVN-PRE on a program patterned off the CFG in figure 1 should produce correct results.

- Demonstrate that this implementation of PRE subsumes GCSE and LICM, as it is supposed to.

  If the GVN-PRE results need to be improved, extending GVN to use LLVM's load value numbering[8] analysis results may be an option to consider. VanDrunen et al's GVN-PRE paper [7] hinted that removal of redundant loads and stores can have a significant impact on the performance of instruction removal optimizations.

In case things turn out tougher than anticipated, I hope to reach one of the two intermediate points, so there is something tangible to demonstrate.

References

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

[2] Tanya Brethour et al. An llvm implementation of ssapre.

[3] J. Knoop, O. Ruething, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 224–234, San Francisco, CA, June 1992.

[4] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

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

[6] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM Press.

[7] Thomas VanDrunen and Antony L. Hosking. Value-based partial redundancy elimination. In *CC, 2004.*

[8] Thomas John VanDrunen. Partial Redundancy Elimination for Global Value Numbering. *PhD thesis, Purdue University, August 2004.*

condition into both the *if* and *else* predecessors.

[7]It uses def-chains to identify lexically identical expressions

[8]From its description in the source, "Load Value numbering uses lexically identical load instructions and uses alias analysis to determine which loads are guaranteed to produce the same value"