CSU498 Project
Report

Global Value Numbering

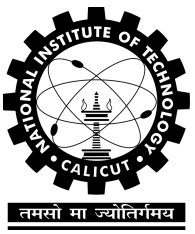
Submitted in partial fulfilment of
the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science and Engineering

Submitted by

Kartik Singhal
B090566CS

Under the guidance of
Dr Vineeth K Paleri



Department of Computer Science and Engineering
National Institute of Technology Calicut
Calicut, Kerala, India - 673 601

Monsoon Semester 2012

Department of Computer Science and Engineering

National Institute of Technology Calicut

Certificate

This is to certify that this is a bonafide record of the project presented by the student whose name is given below during Monsoon Semester 2012 under the course CSU498 Project in partial fulfilment of the requirements of the degree of Bachelor of Technology in Computer Science and Engineering.

Kartik Singhal
B090566CS

Dr Vineeth K Paleri
(Project Guide)

Anu Mary Chacko
(Course Coordinator)

Date:

Abstract

Value numbering is a compiler optimization technique to detect redundancy in expressions. We study it in detail; review and compare known algorithms; and aim to implement one of the best among them, and in the process, improve upon the algorithm if possible. We study GCC as an implementation platform for compiler research and implement constant propagation optimization using a naïve approach to gain practical understanding.

Contents

List of Figures

Chapter 1

Problem Definition

To study global value numbering, a compiler optimization, in detail; to review and compare known algorithms; to implement one of the best among them; and in the process, improve upon the algorithm if possible.

Chapter 2

Introduction

2.1 Background and Recent Research

2.1.1 Global Value Numbering

Global value numbering (GVN) is a program analysis that categorizes expressions in the program that compute the same static value[1]. This information can be used to remove redundant computations.

2.1.2 Literature Survey

Kildall in 1973[2] introduced the most precise global analysis algorithm for program optimization by using the concept of an optimizing function for generalization. Optimizing functions for constant propagation and common subepression elimination were described but the algorithm had an exponential cost.

Alpern, Wegman and Zadeck (AWZ) introduced an efficient algorithm in 1988[3] which uses a value graph to represent symbolic execution of a program. It represents the values of variables after a join using a selection function ϕ, similar to the selection function in static single assignment (SSA) form, and treats these functions as uninterpreted, hence remains incomplete.

The polynomial time algorithm introduced by Ruthing, Knoop and Steffen (RKS) in 1999[4] extends the algorithm by AWZ. It employs a normalization process using some rewrite rules for terms involving ϕ functions, until congruence classes reach a fixed point. This results in discovery of more equivalances and is optimal for acyclic programs but remains incomplete.

Karthik Gargi proposed balanced algorithms in 2002[5] which extend AWZ to perform forward propagation and reassociation and to consider back edges in SSA graph. This discovers more equivalences but is still incomplete.

Gulwani and Necula in 2004[6] proposed a polynomial time algorithm which is optimal if only equalities of bounded size are considered.

We are interested in a sufficiently efficient but complete algorithm for global analysis problem.

2.2 Motivation

A compiler is a fairly large software program and forms an excellent software engineering case study. Optimizing compilers are hard to build especially when software engineering practices encourage generic programming, which is good for code reuse but bad for runtime performance. Study of compiler optimizations provides a good blend of theory (for generality and correctness) and practice (for validation and efficiency). Global Value Numbering, specifically, is an interesting global dataflow analysis for study.

Chapter 3

Work Done

3.1 Compiler Infratructures

There are various compiler infrastructures available as options for experimental compiler research. We looked into some of the most popular ones: SUIF, GCC and LLVM.

SUIF Compiler System[8] is a compiler infrastructure made freely available by the Stanford SUIF Compiler Group to support research in optimizing and parallelizing compilers and uses a program representation, also called SUIF (Stanford University Intermediate Format). It supports C, C++, Fortran and Java as frontends and C, x86, Alpha and MachSUIF as backends. It provides an extensible SUIF representation which allows programmers to extend the representation without requiring updating the infrastruture.

GCC (GNU Compiler Collection)[9] is a compiler system produced by the GNU Project which supports various programming languages. GCC is adopted as the standard compiler by most of the modern UNIX-like operating systems, including Linux and BSD. Frontends are available for C, C++, Objective-C, Objective-C++, Fortran, Java, Ada, and Go. It has been ported to a wide variety of processor architecures and is also available for most embedded platforms.

LLVM[10] is a fairly new compiler infrastructure developed at University of Illinois, which is designed for compile-time, link-time, run-time and idle-time optimizations of programs. It has a language-agnostic design, which allows it to support a wide variety of frontends, some of which include C, C++, Objective-C, D, Fortran and Ada among others. Clang, LLVM frontend for C-like languages, claims to be much faster and to use much less memory than GCC[11]. But LLVM supports fewer targets compared to GCC.

We choose GCC as our implementation platform as it is a popular, pro-

fessional, open-source compiler and could yield more realistic results.

3.2 Structure of GCC

Like most compilers, the compilation process of a GCC-based compiler can be conceptually split up in three phases[12]:

- There is a separate front end for each supported language. A front end takes the source code, translates that source code into a semantically equivalent, language independent abstract syntax tree (AST). The syntax and semantics of this AST are defined by the GIMPLE language, the highest level language independent intermediate representation GCC has.
- This AST is then run through a list of target independent code transformations that take care of constructing a control flow graph, and optimizing the AST for optimizing compilations, lowering to non-strict RTL (expand), and running RTL based optimizations for optimizing compilations. The non-strict RTL is handed over to more low-level passes.
- The low-level passes are the passes that are part of the code generation process. The first job of these passes is to turn the non-strict RTL representation into strict RTL. Other jobs of the strict RTL passes include scheduling, doing peephole optimizations, and emitting the assembly output.

Neither the AST nor the non-strict RTL representations are completely target independent, but the GIMPLE language is target independent.

3.2.1 IR Forms

GCC internally uses several intermediate forms:

- GENERIC
- GIMPLE
- Register Transfer Language (RTL)

GENERIC

GENERIC[13] is an intermediate representation language aimed to provide a language-independent way of representing an entire function in trees. The C, C++ and Java front ends of GCC produce GENERIC directly in the front end. Other front ends instead have different intermediate representations after parsing and convert these to GENERIC.

GIMPLE

GIMPLE[14] is a three-address representation derived from GENERIC by breaking down GENERIC expressions into tuples of no more than 3 operands (with some exceptions like function calls) using temporary variables. All frontends of GCC convert their IR forms to GIMPLE (via GENERIC) for the middle stage of GCC, which does most of the code analysis and optimization.

Register Transfer Language

Register transfer language (RTL)[15] is an intermediate representation (IR) that is very close to assembly language (low-level). In GCC, RTL is generated from the GIMPLE representation, transformed by various passes in the GCC 'middle-end', and then converted to assembly language.

3.2.2 Work Flow

GCC supports plugin mechanism for implementing optimization passes (manipulating GIMPLE form). The plugin structure is shown in figure 3.1. The pass manager takes care of choosing and running all the available passes and their order on the program AST. A plugin could be either a static plugin or a dynamic one.

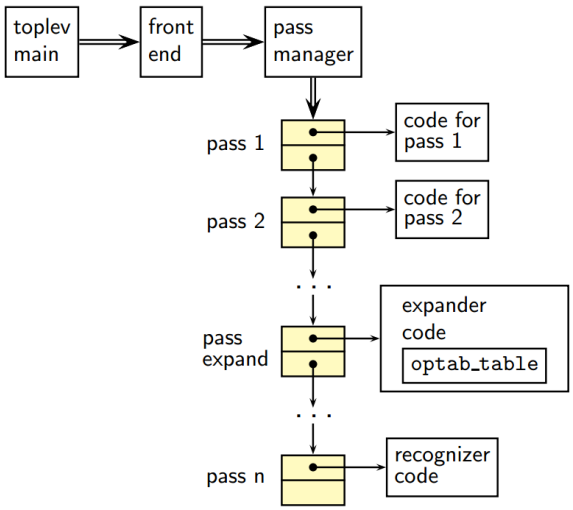


Figure 3.1: Plugin Structure in GCC[7]

We choose to implement our pass as a dynamic plugin since it's easier to integrate it with GCC without recompiling. Figure 3.2 shows how a dynamic plugin can be inserted at a desired point in the list of passes.

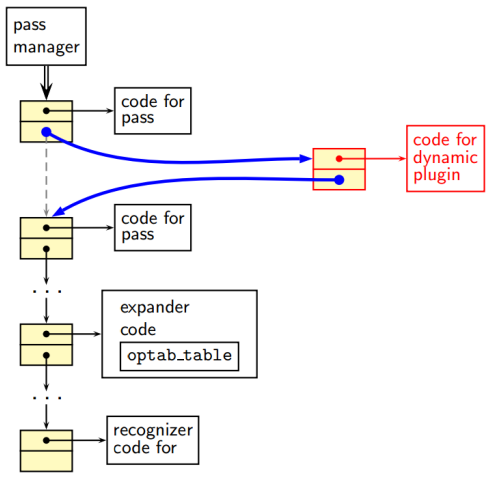


Figure 3.2: Dynamic Plugin Mechanism in GCC[7]

3.3 Constant Propagation - A Naïve Implementation

For gaining familiarity with GCC APIs for GIMPLE modification, a simple plugin was developed for a naïve implementation of constant propagation optimization, which is applicable only for simple linear code. We store the latest value for a constant variable in a data store and replace the use of that variable in following statements with the actual value in a single pass. The code for the plugin is available at `https://github.com/k4rtik/btp-gvn/tree/master/practice/const-prop`. Figure 3.3 shows the test program; figure 3.4 its GIMPLE form as produced by GCC; the intermediate form produced after the CFG pass of GCC is shown in figure 3.5; and lastly the dump produced by our plugin including the converted intermediate code is shown in figure 3.6.

```
int main()
{
        int u = 20;
        int v = 30;
        int w = u + v;
        int x = 10;
        int y = 40;
        int z = x + y;
        int a = y - x;
        int b = x + 5;
        int c = 6 + y;

        return 0;
}
```

Figure 3.3: Test Program in C

```
main ()
{
  int D.1596;
  int u;
  int v;
  int w;
  int x;
  int y;
  int z;
  int a;
  int b;
  int c;

  u = 20;
  v = 30;
  w = u + v;
  x = 10;
  y = 40;
  z = x + y;
  a = y - x;
  b = x + 5;
  c = y + 6;
  D.1596 = 0;
  return D.1596;
}
```

Figure 3.4: In GIMPLE IR form

```
;; Function main (main)

Merging blocks 2 and 3
main ()
{
  int c;
  int b;
  int a;
  int z;
  int y;
  int x;
  int w;
  int v;
  int u;
  int D.1596;

<bb 2>:
  u = 20;
  v = 30;
  w = u + v;
  x = 10;
  y = 40;
  z = x + y;
  a = y - x;
  b = x + 5;
  c = y + 6;
  D.1596 = 0;
  return D.1596;

}
```

Figure 3.5: After CFG pass

```
;; Function main (main)


Constants dump:
==============
u:20, v:30, x:10, y:40,

u = 20;
v = 30;
w = 20 + 30;
x = 10;
y = 40;
z = 10 + 40;
a = 40 - 10;
b = 10 + 5;
c = 40 + 6;
D.1596 = 0;
return D.1596;
```

Figure 3.6: Dump produced by our plugin

Chapter 4

Future Work

We plan to use GCC for implementation of a global value numbering algorithm, test and compare its performance with other known algorithms and look for any possible improvements in the algorithm during the remaining course of the project.

Chapter 5

Conclusion

Pros and cons of a number of known value numbering algorithms have been evaluated and familiarity with functionality of GCC as a compiler research infrastructure gained which will aid in further implementation work as next step in the project.

References

[1] VanDrunen, T. J. 2004. Partial redundancy elimination for global value numbering (Doctoral dissertation, Purdue University).

[2] Gary A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (POPL '73). ACM, 194-206. http://doi.acm.org/10.1145/512927.512945

[3] B. Alpern, M. N. Wegman, and F. K. Zadeck. 1988. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL '88). ACM, 1-11. http://doi.acm.org/10.1145/73560.73561

[4] Rüthing, O., Knoop, J., & Steffen, B. 1999. Detecting equalities of variables: Combining efficiency with precision. *Static Analysis*, 848-848.

[5] Karthik Gargi. 2002. A sparse algorithm for predicated global value numbering. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (PLDI '02). ACM, 45-56. http://doi.acm.org/10.1145/512529.512536

[6] Gulwani, S., & Necula, G. 2004. A polynomial-time algorithm for global value numbering. *Static Analysis*, 703-1020.

[7] Essential Abstractions in GCC '12 - A workshop on GCC Internals by GCC Resource Center, IIT Bombay. http://www.cse.iitb.ac.in/grc/gcc-workshop-12

[8] The SUIF2 Compiler System - http://suif.stanford.edu/suif/suif2/

[9] GCC, the GNU Compiler Collection - http://gcc.gnu.org/

[10] The LLVM Compiler Infrastructure Project - http://llvm.org/

[11] Clang vs Other Open Source Compilers - Clang vs GCC - http://clang.llvm.org/comparison.html#gcc

[12] Structure of GCC - http://gcc.gnu.org/wiki/StructureOfGCC

[13] GENERIC - http://gcc.gnu.org/onlinedocs/gccint/GENERIC.html

[14] GIMPLE - http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

[15] RTL Representation - http://gcc.gnu.org/onlinedocs/gccint/RTL.html