CSU499 Project
Report

Global Value Numbering

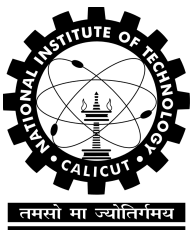
Submitted in partial fulfillment of
the requirements for the award of the degree of

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

Winter Semester 2013

Department of Computer Science and Engineering

National Institute of Technology Calicut

Certificate

This is to certify that the project work entitled "**Global Value Numbering**", submitted by KARTIK SINGHAL (B090566CS) to National Institute of Technology Calicut towards partial fulfillment of the requirements of the award of degree of Bachelor of Technology in Computer Science and Engineering is a bonafide record of the work carried out by him during academic year 2012-13 under my supervision and guidance.

Dr Vineeth K Paleri
(Project Guide)

Place: Calicut
Date: May 6, 2013

Anu Mary Chacko
(Course Coordinator)

Abstract

Value numbering is a compiler optimization technique to detect redundancy in expressions. We study it in detail, review and compare known algorithms, and partially implement one of the best among them. We study GCC as an implementation platform for compiler research and use it to implement the chosen algorithm 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 subexpression 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 equivalences 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.

A Simple Algorithm for Global Value Numbering proposed recently by N Saleena and Vineeth Paleri[7] builds on Kildall's approach and is complete in terms of number of redundancies identified. It introduces the concept of a *Value Expression* to represent a set of equivalent expressions which makes this algorithm fairly simple to understand. We decided to implement this algorithm on GCC framework as we are interested in a 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. 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 data flow analysis for study. Also, this was an opportunity to work on a popular open source software such as GCC, learn about its huge professional code base and interact with its community of developers.

Chapter 3

Work Done

3.1 Compiler Infrastructures

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 front ends and C, x86, Alpha and MachSUIF as back ends. It provides an extensible SUIF representation which allows programmers to extend the representation without requiring updating the infrastructure.

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. front ends are available for C, C++, Objective-C, Objective-C++, Fortran, Java, Ada, and Go. It has been ported to a wide variety of processor architectures 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 front ends, some of which include C, C++, Objective-C, D, Fortran and Ada among others. Clang, LLVM front end 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, professional, 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. Compiler pipeline for GCC can be seen in figure 3.1.

3.2.1 IR Forms

GCC internally uses several intermediate forms:

- GENERIC
- GIMPLE
- Register Transfer Language (RTL)

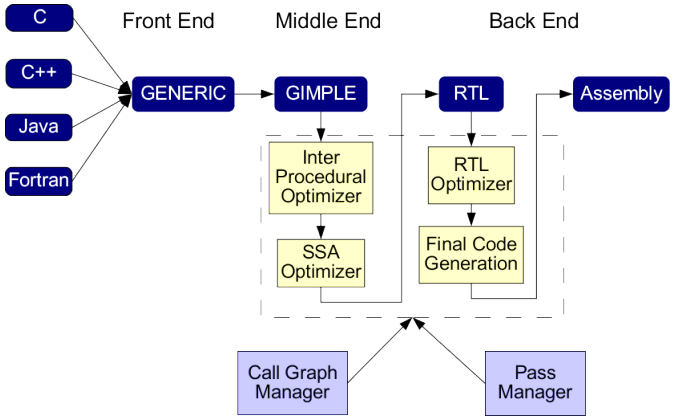


Figure 3.1: Compiler Pipeline[13]

GENERIC

GENERIC[14] 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[15] 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 front ends 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)[16] 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 GCC Plugin Mechanism

GCC supports plugin mechanism for implementing optimization passes (manipulating GIMPLE form). 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.

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.

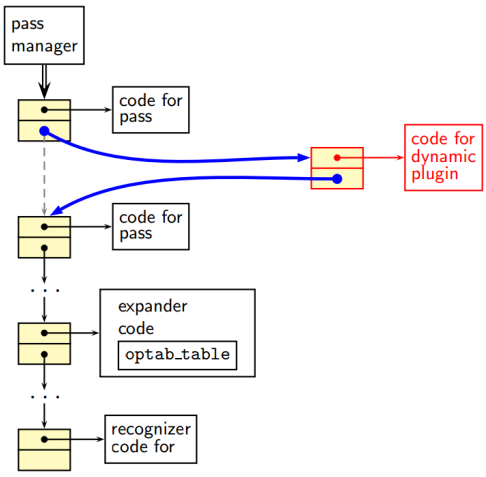


Figure 3.2: Dynamic Plugin Mechanism in GCC[17]

3.3 Implementation

An extensive study of GCC Application Programming Interface for implementation of optimization passes was done using GCC source code, GCC Internals Documentation[18], slides from GCC Internals Course[13] and those from IITB GCC Resource Center's workshop – Essential Abstractions in GCC[17].

3.3.1 Plugin

A plugin was developed for the chosen algorithm – Simple Algorithm for Global Value Numbering[7] – focusing on verifying the correctness without regard to efficiency. The current implementation can handles only linear code. Expression pools at all program points are maintained in a custom data structure and transfer function applied over them iteratively as described in the algorithm[7]. The code for the plugin is available at `https://github.com/k4rtik/btp-gvn/tree/master/gvn-plugin`. The pass implemented in this plugin runs just after the *cfg* pass of GCC.

Figure 3.3 shows the test program; the intermediate form (GIMPLE) produced after the CFG pass of GCC is shown in figure 3.4; and the dump produced by our plugin is shown in figure 3.5.

```c
int main()
{
        int b = 2, c = 3, d = 4;
        int a = b + c;
        b = a - d;
        int e = b + c; // not redundant
        int f = a - d; // redundant
        return 0;
}
```

Figure 3.3: Test Program in C

```
;; Function main (main)

Merging blocks 2 and 3
main ()
{
  int f;
  int e;
  int a;
  int d;
  int c;
  int b;
  int D.1593;

<bb 2>:
  b = 2;
  c = 3;
  d = 4;
  a = b + c;
  b = a - d;
  e = b + c;
  f = a - d;
  D.1593 = 0;
  return D.1593;

}
```

Figure 3.4: After CFG pass

3.3.2 Challenges Faced

- GCC is a long running (about 27 years!), production quality framework (in terms of completeness and usefulness), and consists of over 4 millions LOC (lines of code) spread over more than 35,000 files[17]; which is a fairly difficult code base to understand. Attending the IITB workshop[17] in summer 2012 and revisiting classics like Kernighan & Ritchie[19] helped to understand and navigate through the source code.

- The GCC Internals Documentation[18], although about 700 pages in length, is considered quite sparse in terms of completeness. Getting help over GCC mailing list[20] and GCC development IRC channel[21], however slow in response at times, was a good experience.

```
;; Function main (main)

b not found!

Add a new class:
================
Class 0 -> v.1 -> b -> 2

c not found!

Add a new class:
================
Class 0 -> v.1 -> b -> 2
Class 1 -> v.2 -> c -> 3

d not found!

Add a new class:
================
Class 0 -> v.1 -> b -> 2
Class 1 -> v.2 -> c -> 3
Class 2 -> v.3 -> d -> 4

a not found!

Add a new class:
================
Class 0 -> v.1 -> b -> 2
Class 1 -> v.2 -> c -> 3
Class 2 -> v.3 -> d -> 4
Class 3 -> v.4 -> a -> v.1 + v.2

// Continued on right
```

```
// Continued from left

b found!

Delete from class:
==================
Class 0 -> v.1 -> 2
Class 1 -> v.2 -> c -> 3
Class 2 -> v.3 -> d -> 4
Class 3 -> v.4 -> a -> v.1 + v.2

Add a new class:
================
Class 0 -> v.1 -> 2
Class 1 -> v.2 -> c -> 3
Class 2 -> v.3 -> d -> 4
Class 3 -> v.4 -> a -> v.1 + v.2
Class 4 -> v.5 -> b -> v.4 - v.3

e not found!

Add a new class:
================
Class 0 -> v.1 -> 2
Class 1 -> v.2 -> c -> 3
Class 2 -> v.3 -> d -> 4
Class 3 -> v.4 -> a -> v.1 + v.2
Class 4 -> v.5 -> b -> v.4 - v.3
Class 5 -> v.6 -> e -> v.5 + v.2

f not found!

Insert into class:
==================
Class 0 -> v.1 -> 2
Class 1 -> v.2 -> c -> 3
Class 2 -> v.3 -> d -> 4
Class 3 -> v.4 -> a -> v.1 + v.2
Class 4 -> v.5 -> b -> v.4 - v.3 -> f
Class 5 -> v.6 -> e -> v.5 + v.2
```

Figure 3.5: Dump produced by our plugin

Chapter 4

Future Work

The current implementation handles only linear code, implementation of confluence operation to handle branching and looping is a definite work for future. Apart from this, as previously planned, experimental comparison with other algorithms can be taken up.

We used a slightly older version of GCC viz. 4.6.3 since GCC was moving to C++ as the base language in version 4.7, the current version is 4.8 and the plugin implementation can be ported for more recent versions.

We did not consider the performance of the implementation for the sake of time and focus on correctness. A quality implementation can be done and even submitted as a patch to GCC community.

Chapter 5

Conclusion

Value numbering is studied in detail and multiple known algorithms for global analysis evaluated for completeness. Familiarity with functionality of GCC as a compiler research infrastructure gained including some understanding of GCC source code. One of the algorithms is implemented as a dynamic plugin for linear code.

The complete goals of the project were not met due to over-optimistic estimate of time required for implementation. Lack of documentation and some difficulty faced in getting timely help added to the delay.

Acknowledgments

I would like to express my deep gratitude to my guide **Dr Vineeth Paleri**, Professor, CSED for his invaluable help and guidance during the course of this project, and earlier for the recommendation to visit IIT Bombay for the GCC workshop which led to this project.

Many people from GCC Resource Center at IIT Bombay whom I met during the GCC workshop, especially **Prof Uday Khedkar** and **Swati Rathi**, PhD student at IITB, both of whom very patiently answered many of my emails regarding understanding of GCC concepts and code base. Prof Uday deserves special mention for allowing me to attend his workshop without charge and hence, making me promise to do a project in compilers.

Special thanks are due to my course coordinator **Ms Anu Mary Chacko**, Assistance Professor, CSED for her constant support and reschedulings when I faced difficulty with the evaluation schedules.

Kartik Singhal

May 2013
National Institute of Technology Calicut

References

[1] VanDrunen, T. J. 2004. Partial redundancy elimination for global value numbering (Doctoral dissertation, Purdue University). http://docs.lib.purdue.edu/dissertations/AAI3154748/

[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. http://dl.acm.org/citation.cfm?id=718137

[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. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.7271

[7] Saleena Nabeezath, Vineeth Paleri. 2013. A Simple Algorithm for Global Value Numbering. *arXiv:1303.1880* – http://arxiv.org/abs/1303.1880

[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] Compiler Pipeline, GCC Internals Course – November 2007 – http://www.airs.com/dnovillo/200711-GCC-Internals/

[14] GENERIC – http://gcc.gnu.org/onlinedocs/gccint/GENERIC.html

[15] GIMPLE – http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

[16] RTL Representation – http://gcc.gnu.org/onlinedocs/gccint/RTL.html

[17] 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

[18] GCC Internals Documentation (for GCC 4.6.3) – http://gcc.gnu.org/onlinedocs/gcc-4.6.3/gccint/

[19] The C Programming Language, Second Edition by Kernighan & Ritchie – http://cm.bell-labs.com/cm/cs/cbook/

[20] The gcc mailing list archives – http://gcc.gnu.org/ml/gcc/

[21] GCConIRC – http://gcc.gnu.org/wiki/GCConIRC