Mårten Kongstad

An Implementation of Global Value Numbering in the GNU Compiler Collection with Performance Measurements

To my family, that they may learn my secret doings during the past few months

Abstract

During the first half of 2004, the GNU Compiler Collection (GCC) underwent a major internal transition, as the development branch responsible for the Static Single Assignment (SSA) form — a clever scheme for intermediate code representation — was incorporated into GCC's main branch. Many compiler optimisations that existed pre-SSA benefit greatly from translation to SSA form; such is the case of Global Value Numbering (GVN), which is an algorithm for finding and removing redundancies in the form of expressions calculated more than once. This paper presents the GVN optimisation and how the author implemented it on SSA form for use in GCC, accompanied by GVN performance measurements derived from the SPEC CPU2000 benchmark suite. Performance was boosted by up to 6.1 %: these results motivate the inclusion of GVN on SSA form in GCC.

Acknowledgements

Several people have all had quite a notable influence on this work. For this the author is most apprechiative, and he would like to express his deepest gratitude to the following wonderful individuals.

My supervisor, Dr. Jonas Skeppstedt, has throughout the project shown unparallelled enthusiasm for my progress; also, his extended knowledge on optimising compilers have proven to be a virtual treasure trove, and it is with great pleasure I continue to revel in this facinating topic long after this project has come to an end.

Throughout my education an oracle of all things computer-related, Christian Andersson's assistance and general interest during this project substantially boosted my confidence.

The familiarity and close correspondance with fellow student Niclas Paulsson have provided many prolific discussions and a means of conquering possible frustration.

The acquaintance with Henrik Strand has given me quickly delivered proofreadings and an excellent exchange of thoughts and ideas.

Finally, a very special thanks goes to my girlfriend Elin Brun who, despite me being, at times, totally immersed in this project, succeeded in providing linguistic aid as well as, on a more personal note, maintaining all those traits of hers, that make her my Love.

Mårten Kongstad

September 2004

Contents

Chapter 1

Introduction

"How now! What art thou? Whence comest? What wantest?"
THE DWARF in William Morris' The Wood Beyond the World

1.1 Background

A most critical issue in software development is the choice of compiler. Characteristics to look for, apart from the obvious requirement of the compiler generating correct code from the user's input, are various optimisations on the output. By applying apt transformations, the compiler may deliver code that runs in less clock cycles (and occasionally, takes up less space), than should it, had the compiler not performed these algorithms. The time spent on optimisations during compilation is often gained multiple times during execution.

The GNU Compiler Collection (GCC) is a compiler for several popular platforms, including Linux, Solaris and Windows. Providing front-ends for numerous computer languages, such as C, C++ and Java, GCC is utilised by business enterprises, universities and private persons alike. It is distributed under the GNU General Public Licence, which, in short, means that GCC is free and ships with its source code, available for anyone to modify. Currently, GCC undergoes a major change internally, as it is altered to make use of a relatively new scheme for optimising compilers, the Static Single Assignment (SSA) scheme.

There are many optimisations that make use of SSA, and more are invented on a steady basis. Only in its SSA-infancy, GCC implements a select few SSA-optimisations: a set of algorithms that does not yet include the Global Value Numbering (GVN) optimisation. This specific algorithm removes redundant expressions that compute the same value as some preceding expression, and instead makes use of the already obtained result directly rather than repeating a possibly expensive operation.

1.2 Notes to the reader

1.2.1 Notation

The notation used in this work needs some brief comments. Firstly, in the pseudocode listings, we have chosen to use a more C like syntax than that

Figure 1.1: The gnu head, used to highlight paragraphs specific to the GNU project or GCC.

employed by many other works on computer code. This is motivated by its clarity, and should not be of any concern to the reader.

Secondly, some paragraphs are marked with one of the GNU project's logotypes, a gnu head (see Figure 1.1). These parts of the text deal specifically with the GNU project, or more commonly, with GCC. They may be viewed as advanced topics, reminiscent of Knuth's *dangerous bends* in [7], and may be skipped during a first read without loss of the general picture.

1.2.2 On Internet references

Since the contents of the Internet change relatively frequently, there is no guarantee that the Internet references in this document still hold the same material after this work has been printed as they did at the time of this writing. The author has intentionally chosen to cite Internet addresses that are likely to remain the same, or at least, cover the same topic, during the foreseeable future. All Internet addresses in this document were found to be correct on August 16, 2004.

1.2.3 The structure of this work

This paper discusses the implementation of GVN on SSA form in GCC. In Chapter 2 an introduction to optimising compilers is given and some definitions are introduced, followed by a presentation of the GCC compiler in Chapter 3. Preceded by a theoretical discussion of the GVN optimisation in Chapter 4, Chapter 5 merges the three preceding parts into an account of the author's implementation of GVN in GCC. Finally, the improvements derived from GVN are presented in the penultimate Chapter 6, with conclusions of this work together with suggestions for future work in the final Chapter 7.

Chapter 2

Optimising compilers and the Static Single Assignment form

To my unutterable amazement, I now, for the first time, saw [...] a performance that struck me as very extraordinary, but which made no impression on anybody else, and with which I soon became as familiar as the rest.

Pip in Charles Dickens' Great Expectations

2.1 Introduction

The main goal of a compiler is to correctly translate the intention of a programmer, expressed in some high level language such as C, to a concrete list of assembler instructions for some specific architecture. However, passing this criteria, for a compiler to be more useful as a programming tool, it should also perform various optimisations on the code in question in order to increase the run speed of the final application (or, more rarely, decrease its memory space usage). Finally, for a compiler to rise above average, it should carry out its tasks as quickly as possible.

2.2 From source code to executable binary

As stated above, the main goal of a compiler is to render high level source code into assembly language. This is not the same as producing an executable binary file, but only the first step towards one. Other tools follow the compiler in the build chain, namely the assembler and the linker (also referred to as the loader). See Figure 2.1 for a schematic overview of these steps.

For a lengthier introduction to compilers in general, see [1].

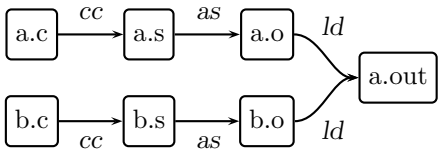

Figure 2.1: Passes from source code to binary. The *cc* program is the C compiler, *as* the assembler, and *ld* the linker.

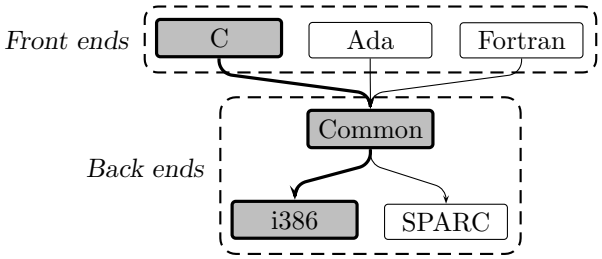

Figure 2.2: Front and back ends of a compiler. The modules activated when compiling C code for the i386 architecture have been highlighted.

When running GCC, the compiler invokes the assembler and linker behind the user's back. Passing the `-v` option to GCC, we may inspect the complex calls performed (the following taken from the author's private system).

```
gcc -v a.c
...
as -V -Qy -o /tmp/ccvi2jWN.o /tmp/cc2cgK3H.s
...
/usr/lib/gcc-lib/i486-linux/3.3.4/collect2 --eh-frame-hdr -m elf_i386
   -dynamic-linker /lib/ld-linux.so.2 -o main /usr/lib/crt1.o
   /usr/lib/crti.o /usr/lib/crtbegin.o
   -L/usr/lib/gcc-lib/i486-linux/3.3.4 -L/usr/lib/ /tmp/ccvi2jWN.o
   -lgcc -lgcc_eh -lc -lgcc -lgcc_eh
   /usr/lib/gcc-lib/i486-linux/3.3.4/crtend.o r/lib/crtn.o
```

Clearly, these command lines are nothing a user writes on the fly, and they are best left to GCC.

2.2.1 The internal structure of a compiler

Virtually all compilers follow a firmly established design strategy, namely that of front and back ends. The idea is to modularise the compiler into building blocks, each responsible for functions specific to either source code (input) or hardware (output). As shown in Figure 2.2, different blocks are then used depending on what source code is compiled, and for what architecture: to compile a source code file, the compiler activates its front end for the programming language in question, and its back end for the target architecture. Traditionally, no other partitioning of the compiler exists. However, it is very likely that some parts of the code, such as the intermediate code representation, may be shared among the back ends and thus separated into a unique module. This part of the compiler, in this work named the Common module, is activated during all successful runs, regardless of source code or platform target.

Apart from the inherent beauty of modularisation itself, this strategy is motivated by greatly easing the task of porting a compiler to a different platform, or adding support for a new programming language, as such work only involves modifying or adding a module, rather than the entire compiler.

One example of the success of front and back ends is the GCC compiler. This tool also incorporates the idea of a Common module in the form of its GENERIC and GIMPLE intermediate code representations (see Section 3.3.2).

2.3 Intermediate code

We now turn our attention to the core of the compiler, i.e. the internal code representation, where most of the optimisations occur. For the discussion below, however, we need the following definition.

DEFINITION 2.1 (THREE OPERAND CODE) *Code in which statements have no more than one destination operand and a maximum of two source operands is said to be three operand code.*

This definition implies that to translate the statement `a = x + y * z` to three operand code, temporary variables must be introduced.

EXAMPLE 2.1 (THREE OPERAND CODE) The equivalent three operand code for the statement `a = x + y * z` is

$T_0 = y * z$
$a = x + T_0$

△

Three operand code also implies that statements with side effects, such as `a = ++x` must be split in two.

We are now ready to examine the compiler core more closely: the output of a front end, and at the same time the input to the Common module, is a description of the source program in a language specific to the compiler. Experience shows that it is advantageous for optimisations and other code transformations if this so called *intermediate code* uses a three operand scheme. Furthermore, the code is divided into logical parts called *basic blocks*, described next.

2.3.1 Basic blocks

DEFINITION 2.2 (BASIC BLOCK) *A* basic block *is the maximal (ordered) sequence of instructions I_i that can only be entered at the first instruction I_0, and exited at the last instruction I_{n-1}.*

The definition implies that basic blocks begin at branch targets (jump labels), function headers, etc, and end at branch instructions (jumps), function footers, etc. It is common to enumerate the blocks and depict them as a graph, with nodes corresponding to the blocks, and edges to possible control flow paths.

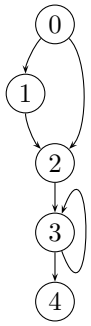

Figure 2.3: The basic blocks deriving from `f()` in Example 2.2 illustrated as a graph.

EXAMPLE 2.2 (BASIC BLOCKS) Dividing

```
int f() {                /* bb0 */
    ...
    if (...) {           /* bb1 */
        ...
    }
    ...                  /* bb2 */
    do {                 /* bb3 */
        ...
    } while (...);
    return ...           /* bb4 */
}
```

into basic blocks, we obtain the structure presented in Figure 2.3. △

2.3.2 Dominance

An often used property of basic blocks (or rather, any node in a directed graph), is that of *dominance*. It is defined as follows.

DEFINITION 2.3 (DOMINANCE) *Consider two nodes v and w in a directed graph. If every path from the start node s to w includes v, then v is said to* dominate *w. We denote this as $v \geq w$.*

If $v \geq w$ and $v \neq w$, v is said to strictly dominate *w. We denote this as $v \gg w$.*

It follows from the definition that the start node s dominates every other node in the graph.

EXAMPLE 2.3 (DOMINANCE (EXAMPLE 2.2 CONTD.)) Revisiting the graph in Figure 2.3, we find that basic block 2 dominates blocks 2, 3 and 4. Basic block 1 dominates no blocks except itself, while block 0 dominates every block in the graph. △

As we will see, the concept of dominance is crucial to the SSA form.

2.3.3 The Static Single Assignment form

Many compiler optimisations, such as Constant Propagation with Conditional Branches [13] and Dead Code Elimination [11], benefit from the same information, in this specific case that of what statement defined a certain variable. This information may be provided by def-use-chains, but also more effectively by putting the intermediate three operand code into a special form, described below. The common request of the optimisations may be exploited by the compiler Common module by putting the intermediate code into this special form, passing it to the optimisations involved, and finally translating it back to "normal" intermediate code.

We are now ready to introduce

Definition 2.4 (The Single Static Assignment form) *A program is said to be on Static Single Assignment form (SSA), if and only if for each variable, that variable is defined at most once, and that definition dominates every use of said variable.*

Naturally, variables tend to be reused and very few programs are initially on SSA form. But by cleverly introducing new variables instead of overwriting old ones, SSA form is achieved, as illustrated by the following example.

Example 2.4 (Trivial SSA conversion) Translating

$x = 1;$
$x = x + 4;$
$y = x;$
$x = y + x;$

into SSA yields

$x_0 = 1;$
$x_1 = x_0 + 4;$
$y_0 = x_1;$
$x_2 = y_0 + x_1;$

△

Straight line code, as in the previous example, is trivially transformed to SSA form by assigning a variable x indexes in the form of running numbers to any definition of x, and by always using the latest index in a use of x. But consider the following code snippet.

```
int f() {
   int x = ...      /* SSA name: x_a */
   ...
   if (...) {
      x = ...       /* SSA name: x_b */
   }
   return x;        /* SSA name: x_? */
}
```

What index should the $x_?$ used by the return-statement have? Depending on whether the if-clause is executed, different indexes are defined at the latest, but which index that actually is, is unknown at compile time. The only statically determinable information is the set of index candidates, namely $S = \{x_a, x_b\}$.

The above problem may be formulated more generally as "when using a variable in SSA form in a basic block reachable via two or more paths that each introduce a new definition of said variable, what index should be applied?" The solution is the introduction of special (artificial) statements called ϕ-functions (or alternatively, ϕ-nodes). These functions take as argument a set of possible indexes for the variable (the set S above) and returns the correct index, i.e. the latest definition. An alternative description of ϕ-functions is that

> ϕ-nodes mean "one of these operands. I don't know which." [12]

Now armed with ϕ-functions, we face another problem: where to place them? Naturally, ϕ-nodes could be introduced in every basic block that joins multiple control flow paths, but such an approach would be both prohibitively expensive and inelegant. It can be shown, however, that for variable x, it suffices to insert ϕ-functions in basic blocks belonging to the so called *dominance frontier* of the block that introduces x, in the dominance frontier of those blocks, etc. An exhaustive discussion on the topic of dominance frontiers falls outside the scope of this work, and the interested reader is referred to e.g. [11, 4]: we are content by presenting the definition of dominance frontiers and an accompanying example. Loosely put, the dominance frontier of a basic block w may be thought of as the basic blocks B_i where the dominance of w has just ended (as illustrated by Example 2.5).

DEFINITION 2.5 (DOMINANCE FRONTIER) *The* dominance frontier DF *of a node* w *is defined as*

$$DF(w) = \{v : \exists\ p \in pred(v), w \geq p \wedge w \not\geq v\}$$

Note that the definition implies that the dominance frontier of the entry node E is the empty set, $DF(E) = \emptyset$, since that node dominates every other in the control flow graph.

EXAMPLE 2.5 (DOMINANCE FRONTIER) In the graph in Figure 2.4, the dominance frontier for node 4 has been highlighted. Every node's dominance frontier is presented in the following table:

	0	1	2	3	4	5
DF	$\emptyset$	{2, 3}	{3}	$\emptyset$	{2}	{2}

△

We are now able to correctly translate complex flow control structures such as loops into SSA form.

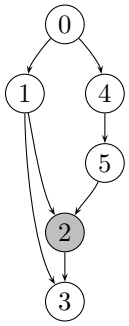

Figure 2.4: The graph used in Example 2.5. The dominance frontier of node 4, $DF(4)$, has been highlighted. It happens that $DF(4) = DF(5)$.

EXAMPLE 2.6 (PLACEMENT OF ϕ-FUNCTIONS) The intermediate code generated from

```
int f(int a, int b)
{
   int x, y;

   x = 1;
   y = 1;
   do {
      a = a + b;
      x = x + a;
      y = y + a;
   } while (a > 0);
   return x + y;
}
```

and translated into SSA is

```
        x0 = 1                      /* bb0 */
        y0 = 1

L0:     a1 = φ1(a0, a2)             /* bb1 */
        x1 = φ1(x0, x2)
        y1 = φ1(y0, y2)
        a2 = a1 + b0
        x2 = x1 + a2
        y2 = y1 + a2
        if (a2 > 0) goto L0; else goto L1

L1:     return x2 + y2              /* bb2 */
```

△

Chapter 3

The GNU Compiler Collection

In my experience, a major problem with GCC *— unless you've been working on it full time for at least five years — is that it's so complex that it's nearly impossible to get started.*

THE GCC MAILING LIST

3.1 Introduction

Embarking on its mission in 1987, the GNU Compiler Collection (henceforth referred to as GCC) successfully continues to "foster a world-class optimizing [sic] compiler" [6] by means of the Open Source Community and the GNU project. Capable of compiling code written in C and its derivatives, Java, and several other languages, and producing assembler code for most of today's popular architectures, including i386 and MIPS,[1] GCC is available to both private persons and companies for free, and usually ships with system distributions supporting GNU.

3.2 Retrieving the compiler

GCC is released under the GNU General Public License (GPL). Software adhering to the GPL are available to the public free of charge, together with its source code. Anyone, including companies, have the right to modify the source code. The GPL states that software under said licence must not be sold, must ship with its source, and that derivatives of the software must be released under the GPL as well. (This does not imply that monetary profit from systems derived from GPL'ed software is void, however. Many large corporations, most notably IBM, make considerable investments in Open Source.) [5]

The GCC sources are accessible via anonymous usage of the Concurrent Version System. See [2] for a more elaborate description of this system; here we

[1] See http://gcc.gnu.org/frontends.html and http://gcc.gnu.org/backends.html for a complete list

only highlight that GCC, apart from the main-line trunk, is available in many development branches: see section 5.3.1 for the implications on this project.[2]

3.3 Intermediate code representation in GCC

3.3.1 The traditional RTL

Traditionally, GCC's intermediate code representation has been the Register Transfer Language (RTL). As can be seen from Example 3.1, the syntax of RTL is heavily inspired by the Lisp language.

EXAMPLE 3.1 (RTL CODE) The following is an excerpt from the RTL manual, intended to give at taste of the same.

```
(define_attr "memory" "fast, slow"
      (const
       (if_then_else (symbol_ref "TARGET_FAST_MEM")
                     (const_string "fast")
                     (const_string "slow"))))
```

△

Further discussion of RTL falls outside the scope of this work, as our implementation is based on RTL's predecessor GIMPLE. For more information on RTL, the interested reader is referred to [12].

3.3.2 Incorporating SSA: GENERIC and GIMPLE

The RTL intermediate language is, for various reasons discussed in [8], not suited for SSA form. Therefore, two new intermediate languages were introduced: GENERIC and its derivative GIMPLE, which is a subset of the GENERIC language and with stricter rules, e.g. the three operand code property. Front ends must now emit GENERIC code rather than RTL; this code is then transformed into GIMPLE by the Common module. However, RTL has not been totally obsoleted, for although SSA equivalent optimisations replace RTL based ones, some platform specific optimisations are best left as RTL based, and RTL is still the input to the back ends. Therefore, after the GIMPLE phase comes an RTL phase. Figure 3.1 summarises how different front ends hand over control to the Common module via GENERIC: cf. Figure 2.2 on page 4.

Note however that front ends may elect to generate GIMPLE directly: this is currently the case of the C and C derivatives front ends.

EXAMPLE 3.2 (SSA IN GCC (EXAMPLE 2.6 CONTD.)) The function `f()` from Example 2.6 on page 9, when translated to SSA by GCC, is listed in Table 3.1. That code is passed to the optimisations on SSA form to work on. △

[2] See http://gcc.gnu.org/cvs.html for a list of active sub-projects

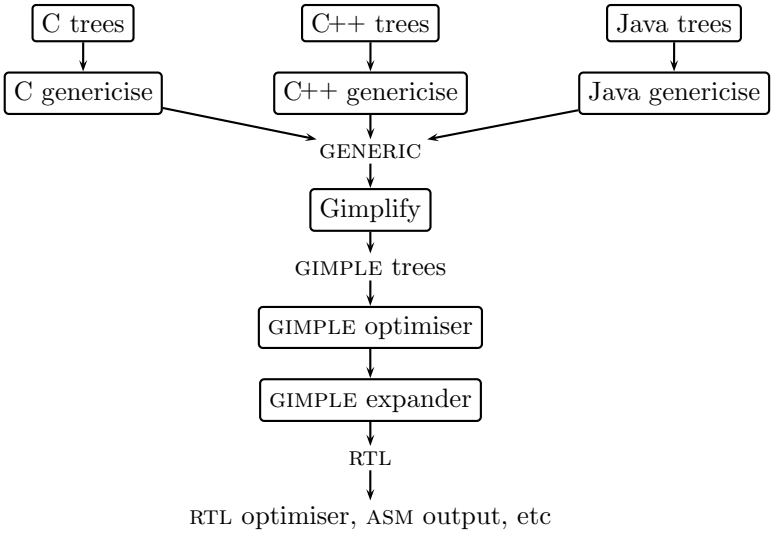

Figure 3.1: The SSA framework in GCC

$x_4 = 1$;
$y_5 = 1$;
<L0>:
a_3 = PHI <$a_6(0)$, $a_8(1)$>;
y_2 = PHI <$y_5(0)$, $y_{10}(1)$>;
x_1 = PHI <$x_4(0)$, $x_9(1)$>;
$a_8 = a_3 + b_7$;
$x_9 = x_1 + a_8$;
$y_{10} = y_2 + a_8$;
if $(a_8 > 0)$ **goto** <L0>; **else goto** <L1>;
<L1>:
return $x_9 + y_{10}$;

Table 3.1: The function `f()` from Example 2.6 as translated to SSA by GCC

Chapter 4

Global Value Numbering

By the time they had diminished from 50 to 8, the other dwarves began to suspect "Hungry". — Gary Larson

4.1 Introduction

Global Value Numbering (GVN) is an optimisation that removes certain redundancies. By considering a set of equivalent expressions, GVN selects a member of the set and uses it consistently wherever any member is used; due to the equivalence, the other expressions may then safely be eliminated.

In this chapter, we will examine GVN more closely. The algorithm is divided into three steps, depicted in Figure 4.1.

4.2 Partitioning

This section details steps 1 and 2 of Figure 4.1. For the discussion below, we need the following definition.

Definition 4.1 (Partition) *A* partition *of a set S of expressions that each introduce a new (SSA) variable, is a set P of disjoint sets B_i, such that $B_i \subseteq S$ for all i, and that $\bigcup_i B_i = S$. The sets B_i are referred to as blocks within the partition P.*

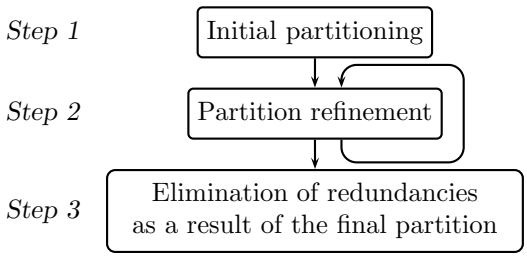

Figure 4.1: Schematic overview of the GVN algorithm

4.2.1 The initial partition

The first step of GVN is to group equivalent expressions in blocks and thus create an initial partition π_0. The GVN algorithm starts optimistically by assuming that all expressions are equivalent, subject to the following constraints:

- global variables and variables passed as arguments to the current function are, due to lack of information, always regarded as unique, and are placed in individual, one-element-only groups,
- constants are grouped depending on their respective values,
- ϕ-functions are grouped based on the basic block they occur in,
- all other expressions are grouped based on their operator exclusively.

In this chapter, we will use the notation ϕ_n to denote a ϕ-function introduced in basic block n. Defining ϕ_n and ϕ_m, where $n \neq m$, as different operators, the above penultimate rule may be merged into the final one.

Cases where the number of arguments to ϕ-functions differ within the same block are handled later by GVN.

The last item ("all other expressions") covers cases such as multiplication. This implies that, e.g., all multiplications in the program initially are seen as equivalent.

In our implementation, we execute Forward Propagation of Single-Use Variables, Copy Renaming as well as Conditional Constant Propagation prior to GVN. Thanks to the former optimisations, we need not concern ourselves with assignments to variables where the right hand side is either a constant or another variable — such statements have all been removed or propagated. We only consider cases where the right hand side is an expression, e.g. additions or ϕ-functions.

EXAMPLE 4.1 (THE INITIAL PARTITION (EXAMPLE 2.6 CONTD.)) Let us consider the function `f()` from Example 2.6 converted to SSA, as shown on page 9 and restated below for the sake of clarity.

$x_0 = 1$ /* bb_0 */
$y_0 = 1$

L_0: $a_1 = \phi_1(a_0, a_2)$ /* bb_1 */
$x_1 = \phi_1(x_0, x_2)$
$y_1 = \phi_1(y_0, y_2)$
$a_2 = a_1 + b_0$
$x_2 = x_1 + a_2$
$y_2 = y_1 + a_2$
if $(a_2 > 0)$ **goto** L_0; **else goto** L_1

L_1: **return** $x_2 + y_2$ /* bb_2 */

Running GVN on the code above, the initial partition π_0 is

$$\pi_0 = \left\{ \begin{array}{l} B_0 : \{x_0 = 1,\ y_0 = 1\} \\ B_1 : \{a_1 = \phi_1(a_0, a_2),\ x_1 = \phi_1(x_0, x_2),\ y_1 = \phi_1(y_0, y_2)\} \\ B_2 : \{a_2 = a_1 + b_0,\ x_2 = x_1 + a_2,\ y_2 = y_1 + a_2,\ retval = x_2 + y_2\} \end{array} \right\}.$$

△

4.2.2 Refining the partition

Obviously, the rough initial partition π_0 is incorrect in virtually all cases, since all operations of the same type, e.g. additions, in a program rarely compute the same value. Therefore, GVN proceeds by iteratively refining the partition. This may be done in one of two ways. Here, we detail the clearer, but slower $\mathcal{O}(n^2)$ variant, and outline its $\mathcal{O}(n \log n)$ counterpart.

The $\mathcal{O}(n^2)$ partitioning scheme

In Table 4.1, the pseudocode for the $\mathcal{O}(n^2)$ partitioning variant is listed. We will now scrutinise this central part of GVN in plain text.

The task of this step of GVN is to create new partitions, each a little closer to the final partition π_f. This is done by selecting a block B_i from the latest partition π_k and placing a member m of B_i in a new block B_x in π_{k+1}. The other members of B_i are then added to

- B_x if the N number of operands match that of m and if, for all $0 \leq j < N$, the expression that introduces operand j is in the same block (in π_k) as the expression that introduces operand j of m, or to
- a dedicated "spill" block B_y in π_{k+1}, otherwise.

(The above logic is implemented in the `match()` subroutine of Table 4.1.) At this stage, π_{k+1} contains one or two blocks. The process repeats itself for all the remaining blocks in π_k, each time adding another one or two blocks to π_{k+1}.

When the new partition π_{k+1} is finished, if $\pi_k = \pi_{k+1}$, GVN has found the final partition, and may continue to step 3 (removing unnecessary statements). If $\pi_k \neq \pi_{k+1}$, a new partition is constructed as detailed above.

It is clear that this iteration will terminate in polynomial time: consider the worst case scenario, where all (the finite) block members go into the spill block, except for m which ends up in a block of its own. Such a process will at some time arrive to a partition π_k where all blocks have only one member. The next partition π_{k+1} will then be equal to π_k and the iteration will terminate. All other possibilities will end sometime before this, with at least one block B_i such that $|B_i| > 1$.

Example 4.2 (Refining the partition (Example 4.1 contd.)) With the initial partition from Example 4.1 as basis, we will now demonstrate the $\mathcal{O}(n^2)$ partitioning. Note that, as the number of partitions mount, we now introduce the notion B_n^m to denote a partitioning block labelled n residing in partition π_m. If it is clear what partition is referred to, we use the old notation.

In Example 4.1 we obtained the initial partition π_0. We now proceed to construct the partition π_1. Starting with an empty set, we create a block B_0^1 to hold $x_0 = 1$ originating from B_0^0. Depending on this expression we now distribute the other expression in B_0^0 to its new location in π_1: since $y_0 = 1$ is an assignment of a constant equal to that of x_0, the expression ends up in B_0^1 together with x_0.

Continuing the construction of π_1, we create a new block B_1^1 to house $a_1 = \phi_1(a_0, a_2)$. Comparing $a_1 = \phi_1(a_0, a_2)$ with $x_1 = \phi_1(x_0, x_2)$, we find, unlike the case with the constants, that the operands are defined in different blocks (x_0 in B_0^0; a_0 is passed as an argument to `f()`, and must therefore be considered a

```
function match(v, w) {
    if (the number of operands of v and w differ) {
        return false
    }
    for (i ← 1; i ≤ the number of operands of v; i ← i + 1) {
        b_v ← the block holding the definition of operand i of v
        b_w ← the block holding the definition of operand i of w
        if (b_v ≠ b_w) {
            return false
        }
    }
    return true
}

function O(n^2)−partition {
    π_0 = {B_0, B_1, ..., B_n}
    i ← 0
    change ← true
    while (change) {
        change ← false
        k ← 0
        foreach (block B_j ∈ π_i) {
            create new block B_k in π_{i+1}
            copy a member v from B_j to B_k
            foreach (member w ∈ B_j : w ≠ v) {
                if (match(v, w)) {
                    add w to B_k
                } else {
                    if (B_{k+1} has not already been created) {
                        change ← true
                        create new block B_{k+1} in π_{i+1}
                    }
                    copy w to B_{k+1}
                }
            }
        }
        k ← k + 1
    }
    i ← i + 1
}
```

Table 4.1: Pseudocode for the $\mathcal{O}(n^2)$ partitioning variant

unique expression, not equivalent to any other, which leads to its placement in a separate block; since this is always true, we simply fall back to this special case every time we find a variable not in any block in a previous partition π_k), and thus, that the two expressions should be placed in separate blocks. Therefore, we create a block B_2^1 to hold $x_1 = \phi_1(x_0, x_2)$, and as we see next, $y_1 = \phi_1(y_0, y_2)$ as well.

History repeats itself when further analysis puts $a_2 = a_1 + b_0$ isolated in its own block B_3^1 and the last three expressions in B_4^1. We now have

$$\pi_1 = \left\{ \begin{array}{l} B_0 : \{x_0 = 1,\, y_0 = 1\} \\ B_1 : \{a_1 = \phi_1(a_0, a_2)\} \\ B_2 : \{x_1 = \phi_1(x_0, x_2),\, y_1 = \phi_1(y_0, y_2)\} \\ B_3 : \{a_2 = a_1 + b_0\} \\ B_4 : \{x_2 = x_1 + a_2,\, y_2 = y_1 + a_2,\, \mathit{retval} = x_2 + y_2\} \end{array} \right\}$$

Since $\pi_0 \neq \pi_1$, we proceed to refine the latter partition into

$$\pi_2 = \left\{ \begin{array}{l} B_0 : \{x_0 = 1,\, y_0 = 1\} \\ B_1 : \{a_1 = \phi_1(a_0, a_2)\} \\ B_2 : \{x_1 = \phi_1(x_0, x_2),\, y_1 = \phi_1(y_0, y_2)\} \\ B_3 : \{a_2 = a_1 + b_0\} \\ B_4 : \{x_2 = x_1 + a_2,\, y_2 = y_1 + a_2\} \\ B_5 : \{\mathit{retval} = x_2 + y_2\} \end{array} \right\}$$

Running step 2 of GVN a third time produces a partition identical to π_2. Thus π_2 is the final partition, and we promote it by renaming it to π_f. △

If, upon inspection, we find that a block B_i only has one member, it obviously cannot be split in two, and thus, is simply copied to the next partition π_{k+1}. This is the approach implemented in the author's GVN algorithm. An alternative would be to move single-member blocks, when found, to a separate set S. When the $\mathcal{O}(n^2)$-iterations come to an end, resulting in a partition π_k, the final partition π_f is the union of the two former sets, i.e. $\pi_f = S \cup \pi_k$.

The $\mathcal{O}(n \log n)$ partitioning scheme

The problem with the $\mathcal{O}(n^2)$ variant is that a single block B_i in partition π_k is split into two blocks in π_{k+1} using every other block in π_k. This is done in $\mathcal{O}(|B_i|)$ time and has to be repeated for every block in the partition.

Amending this shortcoming is the $\mathcal{O}(n \log n)$ variant, based on Hopcroft's partitioning algorithm. Further discussion of this algorithm falls outside the scope of this work; suffice to say that it runs considerably faster than the $\mathcal{O}(n^2)$ counterpart since it approaches the block splitting problem in a more efficient, and in a sense, opposite, way: using one block, the algorithm splits all the other. For more information, the reader is referred to [11].

4.2.3 The final partition

Regardless of what refinement process was chosen, GVN will find the final partition π_f.

```
function dominator_based_gvn {
    foreach (block B_i in π_f) {
        foreach (pair of expressions (v, w) in B_i × B_i : v ≫ w) {
            replace every use of w by a use of v
            remove w
        }
    }
}
```

Table 4.2: Pseudocode for the dominator based replacement algorithm of equivalent expressions

4.3 Removing redundancies

4.3.1 Selecting a representative expression

In step 3 of GVN, we have knowledge of the final partition π_f; in particular, we have knowledge of which statements are equivalent to each other: expressions in the same block are equivalent; blocks with only one member do not contribute to any improvements, and are discarded from further analysis. But the final step of GVN is more complicated than simply choosing one of two equivalent expressions e_a and e_b, say e_a, and use it on all occurrences of e_b, because e_a may not have been introduced at the point of e_b. Consider the following example which highlights this dilemma.

EXAMPLE 4.3 (REPLACEMENT CONSIDERATIONS) In the code snippet

```
if (...) {
    a = x * y;
} else {
    b = x * y;
}
c = x * y;
d = x * y;
```

neither a nor b may replace c, but c may be used instead of d, which can then be safely eliminated. △

Put more formally, an expression e_a may only replace an equivalent expression e_b if e_a dominates e_b. This observation leads to the GVN implementation in Table 4.2. It is, however, possible to safely replace statements such as $c = ...$ in Example 4.3, by using a more complex machinery than the dominator based solution shown here. For more information on how this is done, the interested is referred to [10].

EXAMPLE 4.4 (THE RESULTS OF GVN (EXAMPLE 4.2 CONTD.)) With the information collected in Example 4.2, we may now perform a dominator based redundancy elimination. The final result is presented in Table 4.3. △

Before GVN	**After** GVN
$x_0 = 1$	†
$y_0 = 1$	$y_0 = 1$
L_0: $a_1 = \phi(a_0, a_2)$	L_0: $a_1 = \phi(a_0, a_2)$
$x_1 = \phi(x_0, x_2)$	†
$y_1 = \phi(y_0, y_2)$	! $y_1 = \phi(y_0, x_2)$
$a_2 = a_1 + b_0$	$a_2 = a_1 + b_0$
$x_2 = x_1 + a_2$	! $x_2 = y_1 + a_2$
$y_2 = y_1 + a_2$	†
if $(a_2 > 0)$ **goto** L_0; **else goto** L_1	**if** $(a_2 > 0)$ **goto** L_0; **else goto** L_1
L_1: **return** $x_2 + y_2$	L_1: ! **return** $x_2 + x_2$

Table 4.3: The results of GVN. For the sake of clarity, we restate the original program. A † mark denotes that the line was removed, a ! mark that the line was modified.

4.4 Summary

It is noteworthy that GVN will never insert superfluous statements as a result of heuristics (since there are none), but will only

- remove a statement altogether, or
- replace an operand in an expression, leaving the operator, and hence, the expression's time complexity, intact.

This observation leads to

THEOREM 4.1 (GVN WORST CASE) *The worst case scenario for* GVN *occurs when the algorithm fails to find any expressions viable to modify. The time spent running* GVN *is lost; however, the code will not be bloated with redundant expressions deriving from the optimisation attempt.*

Proof. The theorem follows directly from the discussion above. □

For alternative discourses on GVN, the interested reader is referred to [3, 9].

Chapter 5

Global Value Numbering in The GNU Compiler Collection

Algoritmen är korrekt. ["The algorithm is correct [so it must be your code that is broken]," friendly said in reply to the author's lament for his, at that time, incorrect implementation of the Dead Code Elimination optimisation.]

Christian Andersson

5.1 Introduction

As stated in Section 3.2, GCC is released under the GPL license. This implies that the sources of GCC are extremely easy to obtain. The backside is that during its lengthy progress, GCC has evolved into a complex beast (to say the least) and to create only minor modifications the aspiring programmer must spend much time reading documentation as well as selected, clever parts of the source code. This chapter details the author's attempt at this, which led to a successful implementation of GVN in GCC.

5.2 Adding new optimisations to GCC

The first step when modifying GCC is to obtain the sources, as described in Section 3.2. These files consist of C code files interspersed with clever shell scripts that generate more source code from plain text templates, accompanied by Makefiles for building the compiler.

A quick word about the directory structure of the GCC sources: sub projects are contained in their own subdirectories, but some directories hold shared sources. For our and similar additions it suffices to know that all relevant files reside in the `gcc/` subdirectory — all file names stated here are relative to this location.

What optimisations to perform during a GCC run is controlled by the `-On` command line options, where n may take the values $\{0, 1, 2, 3, s\}$. These options tell GCC to run different presets of optimisations. Higher numbers enable more algorithms, with `-O0` indicating the default, i.e. none. During development it is common to use the `-O2` variant which is a good tradeoff between compilation time and gain; `-O3` is usually reserved for compiling code to be released as it covers the more esoteric algorithms. Of special interest is the `-Os` option, which, unlike the other alternatives, does not optimise for speed, but for code size.

In addition to the `-On` options, individual optimisations may be controlled via options on the form `-f[no-]opt`, where `opt` is the name of a specific optimisation such as `tree-dce` for Dead Code Elimination on SSA form. Thus, optimisations may be enabled or disabled independently, which in reality is a must, since dissimilar code benefit differently from a specific set of optimisations. Indeed, the `-On` sets are heuristics, and experiments with genetic programming have been made to find the optimal set of algorithms to perform on different code pieces.[1]

Naturally, GCC implements a flexible framework to regulate its optimisations internally. Each invocation of an optimisation, called an optimisation pass (note that the same algorithm may be invoked multiple times), is described by an instance of the structure in Figure 5.1. Of particular interest to us are the `gate` and `execute` fields: the former is a function that is queried just prior to starting the optimisation and that returns `true` or `false` depending on whether execution should proceed with the optimisation in question or proceed to the next one; the latter is the optimisation function itself. The relative order of the algorithms is hard coded as a list, and when GCC reaches the optimisation stage, it traverses this list and queries each `gate` function, and, if `true` is returned, executes the optimisation.

To add a new optimisation to GCC, the following files should be introduced or edited. Note the `timevar.def` file, which is a plain text file that is translated to C code by a shell script during the make process. The files, then, are

`tree-ssa-`*opt*`.c` which implements the optimisation *opt*,

`Makefile.in` needs to know about the object file deriving from the previous file,

`timevar.def` is responsible for internal time tracking of optimisations, and must be updated with information about *opt*, as must the two files

`tree-optimize.c` which holds the (ordered) list of optimisations to run, and

`tree-pass.h` which declares the `tree_opt_pass` structure for *opt*.

If the new optimisation should conform to the `-f[no-]opt` option framework, the following files need modification as well (here we find the `common.opt` file, which obeys the same rules as `timevar.def`, described above):

`common.opt` that documents all options, need to know about the new one, as do the header

[1] See http://www.coyotegulch.com/acovea/ for an article on this topic, Scott Robert Ladd's *Acovea: Analysis of Compiler Options via Evolutionary Algorithm*, which dates from March 2004

```
struct tree_opt_pass {
  const char *name;                  /* name used in dump files                */
  bool (*gate) (void);               /* see main text                          */
  void (*execute) (void);            /* the entry point of the optimisation    */
  struct tree_opt_pass *sub;         /* list of sub-passes to run              */
  struct tree_opt_pass *next;        /* optimisation to run after this one     */
  unsigned int static_pass_number;   /* used in dump files                     */
  unsigned int tv_id;                /* used during compiler timing            */

  /* properties input and output from this pass                                */
  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;

  /* bitmask identifying routines to do before and after this pass             */
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
}
```

Figure 5.1: The `tree_opt_pass` structure used to control a designated optimisation pass in GCC

`flags.h` and the two files

`opts.c` and

`toplev.c` that implement the command line option framework.

5.2.1 Basic block and statement iterators

SSA optimisations in GCC work on a function level, i.e. when an optimisation is invoked it is handed a function rather than, say, the entire program or a basic block. To navigate the current code, GCC provides iterators to traverse the basic blocks the function consists of; each basic block holds an iterator to pass over its statements.

From an SSA form point of view, the apparent consistency of iterators is undermined by the fact that ϕ-nodes are handled differently than other statements. Rather than shipping with iterators, a function's ϕ-statements come in a list, which may very well be traversed like the iterator equivalent, but with an interface unique to ϕ-nodes.

It is noteworthy, that throughout the project, ϕ-statements had to be treated differently from other code. Often, the GVN functions begin with an `if` statement to determine whether to execute ϕ-related code or "normal" code.

EXAMPLE 5.1 (GCC ITERATORS) In Table 5.1, code to traverse and print the information fed to an optimisation invocation is listed. Note the GCC specific macros `FOR_EACH_BB` and `TREE_CHAIN` — the compiler is filled with such internal magic. △

```
void print_phis_and_stmts() {
   basic_block bb;
   block_stmt_iterator si;
   tree phi, stmt;

   FOR_EACH_BB (bb) {
      for (phi = phi_nodes(bb); phi; phi = TREE_CHAIN(phi)) {
         print_genetic_stmt(stdout, phi, 0);
      }
      for (si = bsi_start(bb); !bst_end_p(si); bsi_next(&si)) {
         stmt = bsi_stmt(si);
         print_generic_stmt(stdout, stmt, 0);
      }
   }
}
```

Table 5.1: Actual GCC code to traverse all basic blocks fed to an optimisation pass, and for each block print ϕ-nodes and statements on standard out.

5.3 The GVN implementation

5.3.1 The foundation

In the early stages of this project, i.e. in February 2004, the development of GIMPLE and SSA in GCC was contained in a separate development branch. Maturing in this isolation, the so called `tree-ssa-branch` was eventually merged into main-line as of May 13th the same year.

The code in this project was written with the `tree-ssa-branch` as basis, but was trivially adapted to main-line sources after said merge. Thus, the final code is fully compatible with the CVS sources tagged `tree-ssa-post-merge`.

5.3.2 Considerations

In this section, we examine some of the considerations that had to be dealt with during our implementation of GVN. Note that the discussion is, at times, only applicable to our methodology, i.e. the $\mathcal{O}(n^2)$ partitioning scheme.

Commutative expressions

Consider an expression that commute for some operation $\oplus$:

$$x \oplus y = y \oplus x$$

Now assume that in the current partition, x resides in block B_x and that y is in block B_y, such that $B_x \neq B_y$. We wish to consider the two expressions equivalent, but the `match()` function (see Table 4.1 on page 18) will deem them different, based on the order of the operands.

Since other optimisations face the problem with commutative expressions as well, PRE the one closest at hand, a common strategy is to have a pass prior to the optimisations that rearrange the operands of commutative expressions, where applicable. This would lead to, e.g. every $y \oplus x$ to be rewritten as $x \oplus y$. In our implementation, we are content with this, and leave the problem at that.

Where in the list of SSA optimisations should GVN be placed?

With so many other algorithms in the chain of SSA optimisations, where should GVN be inserted? This is a tradeoff between the chance to benefit from simplifications made by earlier optimisations and to provide the same to later ones.

It so happens, that GVN benefits greatly from Constant Propagation (in any incarnation), Copy Renaming, etc. In our case, GVN was placed at the very end of the SSA optimisations, right before the pass that translates the intermediate code from SSA form to RTL, so as to reap said benefits, and minimising the intervention with the established optimisation order.

Chapter 6

Performance measurements

There is now a meta-bug, PR 15502, which references the rest of the known problems which cause the rest of the SPEC CPU2000 Fortran to compile. THE GCC MAILING LIST

6.1 Introduction

When implementing any optimisation, the main question is, naturally, what running speed improvements are achieved. To probe this, one usually measures the time to execute a special test program, called a *benchmark*, once compiled with the optimisation enabled, and once without. In its simplest form, a score can then be computed as the difference between the two times.

It is far from trivial to construct a representative benchmark: it must be thorough, but without overly exercising irrelevant parts of the computer, e.g. the CPU for a graphics card benchmark, or I/O operations for a mathematics library benchmark. Furthermore, to minimise measuring errors, tests should run for relative long times — often many hours.

Supplying a standardised set of benchmarks for consistently measuring the running speed of code on assorted platforms, produced by various compilers, the Standard Performance Evaluation Corporation (SPEC) collects and openly publishes results from said benchmarks — its database currently holds over 2500 entries. This widespread and open mentality motivates our use of the SPEC benchmarks to measure the gains yielded by GVN.

6.2 The SPEC CPU2000 benchmark suite

Common to all the SPEC benchmarks is that they are intentionally geared towards CPU and memory usage, and have a minimised set of I/O operations. The languages used to implement the SPEC benchmarks consist of C, C++, Fortran 77 and Fortran 90. The subset of those tests that we used to measure the effects of GVN, listed in Table 6.1, are all written in C.

Interestingly, one of the SPEC tests is an implementation of the GCC compiler (based on the now stale version 2.7.2.2). Ironically, there exists a well known bug in GCC that prevents the specific test data of this test to execute correctly

Integer benchmarks

Full name	**Category**
164.gzip	compression
175.vpr	FPGA circuit placement and routing
181.mcf	combinatorial optimisation
197.parser	word processing
255.vortex	object-oriented database
256.bzip2	compression
300.twolf	place and route simulator

Floating point benchmarks

Full name	**Category**
177.mesa	3D graphics library
179.art	image recognition/neural networks
182.equake	seismic wave propagation simulation
188.ammp	computational chemistry

Table 6.1: The SPEC tests used to measure the effects of GVN

— the test runs correctly if compiled with most compilers other than GCC. Since we use the latter, however, this specific test was excluded from the benchmark suite.

As is clear from Table 6.1, SPEC benchmarks are named as a number followed by a free text name. Since both must be unique, it is possible to refer to a test by using either, and subsequently we will use the text alternative exclusively. The benchmarks divided into two groups: those tests that exercise integer arithmetic, and those that concentrate on floating point arithmetic.

The result of running a SPEC benchmark is a number of score points for the current configuration — a higher score indicates a faster run, and hence, is better than a lower score.

6.2.1 Runtime options

When running the SPEC benchmarks, the user may specify the number of times a test should run (the score assigned to a test is the mean of the scores from the individual runs), and which of the following three test data sets that should be fed the test:

test very limited test set that runs quickly, intended for testing the SPEC setup

train reasonably sized test set, for use during the development of an optimisation

ref enormous test set, intended for final measurements

For the results of a benchmark run to be statistically comparable, called *reportable*, certain minimal thresholds must be passed.[1] Most notably are the requirements that tests must run at least thrice and with the **ref** input data. These constraints were honoured by all our test runs.

[1] Only reportable results are accepted by SPEC: see http://www.spec.org/cpu2000/results/ for a list of results deriving from over 2500 system configurations

Name	PRE	GVN
full	on	on
std	on	off
repl	off	on
none	off	off

Table 6.2: The four benchmark configurations: all have been compiled with the `-O2` flag, but differ with respect to PRE and GVN

Integer benchmarks

	none	**std**	**repl**	**full**
gzip	94.3	94.6	94.7	94.7
vpr	108	108	108	108
mcf	132	133	133	132
parser	97.1	96.6	96.6	96.4
vortex	128	127	128	125
bzip2	113	111	113	111
twolf	133	130	132	132

Floating point benchmarks

	none	**std**	**repl**	**full**
mesa	70.2	75.2	74.5	76.6
art	209	208	209	208
equake	86.9	87.3	87.0	87.2
ammp	112	107	111	107

Table 6.3: SPEC scores

The benchmark suite was compiled with the `-O2` flag specified, and with the PRE and GVN optimisations individually controlled — see Table 6.2 for our naming convention. This resulted in four sets of results, which we will study next.

6.3 The SPEC results

See Table 6.3 for the SPEC scores of the benchmarks, subject to the four configurations introduced in Table 6.2.

Not all of the benchmarks were possible to optimise with GVN. In fact, only the *parser*, *mesa*, *vortex* and *twolf* tests were susceptible to GVN (as indicated by logging output from the optimisation attempt). The results from the other tests, though not indicative of the impact of GVN, measure another important quantity, namely that of a standard deviation due to error measurement. An example of this are the results of the *ammp* benchmark: since GVN did not affect the test, the difference between the none and repl configurations should be zero, as should the difference between the std and full setups. However, the first pair's scores differ by one. Consulting the other rows of Table 6.3, except for those belonging to the four previously mentioned benchmarks, we find that one is the maximal measurement error. Indeed, it may even be lower, but displays as one, due to rounding errors: cf. the measurement errors of the *gzip* and *equake* tests.

We now turn our attention to the four tests that contained code for GVN to

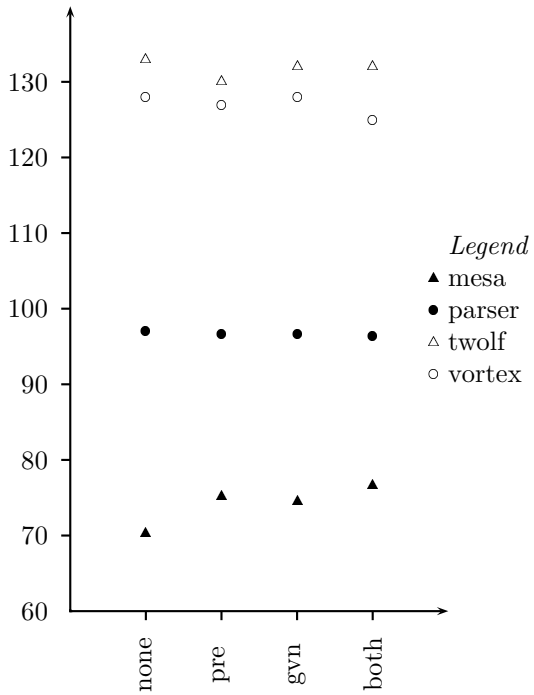

Figure 6.1: The results of the tests susceptible to GVN

edit, the results of which are highlighted in Figure 6.2. The values in this figure are calculated as follows.

$$\textbf{score} = a - b \qquad \% = \frac{a-b}{b} \tag{6.1}$$

where a and b are the scores deriving from the various configurations as indicated below:

$$a : \begin{cases} \textit{repl} & \text{if PRE disabled,} \\ \textit{none} & \text{if PRE enabled.} \end{cases} \qquad b : \begin{cases} \textit{full} & \text{if PRE disabled,} \\ \textit{std} & \text{if PRE enabled.} \end{cases} \tag{6.2}$$

As can be seen in Figure 6.2, some tests produced a lower score with GVN enabled than its counterpart with GVN disabled. However, the negative score fluctuations (save two) all have a magnitude less than one, i.e. the measurement error discussed above, but furthermost, Theorem 4.1 states that GVN will only decrease the number of instructions in a program, if any change is made. Thus, the modifications of these tests must be minor, such as the removal of an initialisation copy, since the improvement does not contribute enough to overcome the measurement error threshold. As is evident from the percentage change, a negative score difference for *any of the tests* does not greatly affect that test's percentage score.

We have now narrowed down our list of benchmarks to *mesa*, which was the only benchmark in our test suite to benefit greatly from GVN. The statements improved upon could have any number of the following characteristics:

	PRE *disabled*		PRE *enabled*	
	score	**%**	**score**	**%**
mesa	4.3	6.1	1.4	1.9
parser	−0.5	−0.5	−0.2	±0.0
vortex	±0.0	±0.0	−2.0	−0.2
twolf	−1.0	−0.1	2.0	0.2

Figure 6.2: Score differences of benchmark runs identical apart from the inclusion of GVN. The values are calculated using Equations (6.1) and (6.2).

- be part of a loop that is executed many times
- be an expensive operation

Another interesting trait of *mesa* is the interaction with GVN and PRE combined. See Section 7.2 on page 35 for a discussion on this topic.

Chapter 7

Conclusions

Please be good enough to put your conclusions and recommendations on one sheet of paper in the very beginning of your report, so I can even consider reading it.

WINSTON CHURCHILL

7.1 Introduction

In this chapter, we review the effort required to implement GVN, the improvements derived thereof, and look ahead at possible future work.

7.2 Comparison with PRE

The Partial Redundancy Elimination (PRE) optimisation is similar to GVN in that they both eliminate redundancies. However, PRE works on a syntactic basis as opposed to GVN's approach based on equivalence. PRE's strategy gives rise not only to code elimination, but also to code motion.

The similarities between the two algorithms mean that they sometimes optimise the same line of code. From the *mesa* benchmark results displayed in Table 6.3 on page 31, we see that both the GVN and the PRE optimisations improve the score individually, but the best score is acquired when both algorithms work in tandem. This benchmark is an excellent example of how the targets of the two optimisations partially overlap; see Figure 7.1.

7.3 Should GVN be incorporated in GCC — future work

The fact that GVN was unsuccessful at finding expressions to eliminate in so many of the SPEC benchmarks is suggestive that most invocations of GVN is a waste of time (however, more tests are needed to consolidate this belief). Furthermore, GVN shares part of its targets with PRE so that the latter may do GVN's work in some cases. These points imply that GVN should not be implemented.

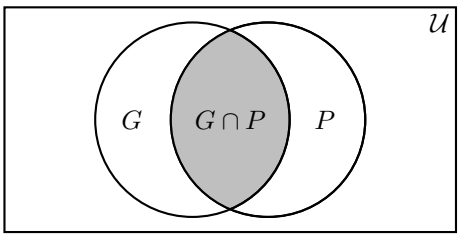

Figure 7.1: The target overlap of GVN and PRE. The set $\mathcal{U}$ is the set of all code being compiled. The sets G and P are the sets of code susceptible to GVN and PRE, respectively. Note that $G \cap P \neq \emptyset$.

However, GVN may well find redundancies overlooked by PRE, and although it might be just a copy statement run once, the code optimised by GVN could also be a costly operation in a loop — the results of the *mesa* benchmark seem to be an example of this. Also, especially for a seasoned GCC programmer, GVN is both easily and quickly implemented. Altogether, these details motivate GVN's incorporation in GCC.

It is the author's opinion, that GVN should be incorporated in GCC but be disabled at the standard optimisation level `-O2`. It should, however, be enabled at higher levels, and should, naturally, provide a `-f[no-]tree-gvn` option. With a clever implementation of the $\mathcal{O}(n \log n)$ partitioning algorithm, the time to execute GVN would probably not stand out among that of the other optimisations.

Appendix A

Program listings

A.1 Introduction

In this appendix we present our implementation of GVN in GCC. To keep the number of pages down, we are satisfied with including only code snippets rather than entire files, except for the main GVN file, `tree-ssa-gvn.c`.

This chapter is divided into two parts. The first deals with the optimisation per se, and the second with the `-f[no-]opt` framework.

A.2 Program listings

A.2.1 The GVN algorithm

tree-ssa-gvn.c

```
/* Global Value Numvering (GVN)
 * Id : tree − ssa − gvn.c, v1.242004/09/2710 : 27 : 51podspeakExp
 */

#include "config.h"
#include "system.h"
#include "coretypes.h"
#include "tm.h"
#include "errors.h"
#include "ggc.h"
#include "tree.h"

#include "rtl.h"
#include "tm_p.h"
#include "hard−reg−set.h"
#include "basic−block.h"

#include "diagnostic.h"
#include "tree−inline.h"
#include "tree−flow.h"
#include "tree−simple.h"
#include "tree−dump.h"
#include "tree−pass.h"
```

```
#include "timevar.h"
#include "expr.h"
#include "flags.h"

#include "assert.h" /* assert() */
#include "sys/times.h" /* times() */

/* macros */

/* if defined, DUMPFILE specifies a file on which to dump debug data;
   if undefined, dump*() become nullops */
/*#define DUMPFILE ("gvn.dump")*/

/* be extra verbose when match()ing; does nothing if DUMPFILE is not
   defined */
/*#define VERBOSEMATCH*/

/* be extra verbosematch when partition()ing; does nothing if DUMPFILE
   is not defined */
/*#define VERBOSEPARTITION*/

/* print statistics on the time taken to run the partitioning and
   replace phases on standard out; this is done independently of
   DUMPFILE */
/*#define VERBOSETIME*/

/* print statistics on the number of replacements and lines erased on
   standard out; this is done independently of DUMPFILE */
/*#define VERBOSEREPLACE*/

#ifdef VERBOSETIME
#define DIFF_IN_MSEC(A, B) \
    (((double) B.tv_sec + (double) B.tv_usec / 1000000.0) - \
     ((double) A.tv_sec + (double) A.tv_usec / 1000000.0))
#endif

#define IS_INITAL_PARTITION_TYPE(T) \
        (T == PLUS_EXPR || \
         T == MINUS_EXPR || \
         T == MULT_EXPR || \
         T == TRUNC_DIV_EXPR || \
         T == CEIL_DIV_EXPR || \
         T == FLOOR_DIV_EXPR || \
         T == ROUND_DIV_EXPR || \
         T == TRUNC_MOD_EXPR || \
         T == CEIL_MOD_EXPR || \
         T == FLOOR_MOD_EXPR || \
         T == ROUND_MOD_EXPR || \
         T == RDIV_EXPR   || \
         T == EXACT_DIV_EXPR || \
         T == MIN_EXPR   || \
         T == MAX_EXPR   || \
         T == LSHIFT_EXPR  || \
         T == RSHIFT_EXPR  || \
```

```
          T == LROTATE_EXPR  || \
          T == RROTATE_EXPR  || \
          T == BIT_IOR_EXPR  || \
          T == BIT_XOR_EXPR  || \
          T == BIT_AND_EXPR  || \
          T == COMPLEX_EXPR)

/* structures */
typedef struct block_expr_t block_expr;
struct block_expr_t {
  tree expr; /* the 'x_0 + y_1' part in 'x_1 = x_0 + y_1' */
  tree defines; /* the 'x_1' part in 'x_1 = x_0 + y_1' */
  tree stmt;
  block_expr* next;
  int removed;
  basic_block bb;
};

typedef struct block_t block;
struct block_t {
  block_expr* expr; /* block expressions in this block */
  block* next; /* next block in the partition this block is part of */
};

/* global variables */
#ifdef DUMPFILE
static FILE* fd = 0;
#endif
#ifdef VERBOSEREPLACE
static unsigned int stmts_modified = 0;
static unsigned int stmts_removed = 0;
#endif
#ifdef VERBOSETIME
static double part_tot = 0;
static double gvn_tot = 0;
#endif
static block* gvn_partition = 0;
static block* gvn_const_partition = 0;

/* function declarations */

/* sets everything up to begin gvn */
static void initialize(void);

/* deallocates resources and performs related work */
static void finalize(void);

/* returns TRUE if all operand definitions i for tree v are in the
   same blocks as the corresponding operand definitions j for tree
   w. Both trees are in partition p */
static int match(tree v, tree w, block* p);

/* partitions the blocks constructed by successive calls to
   add_to_inital_partition(); see Skeppstedt */
```

```
static void partition_n_square(void);

/* entry point for the gvn optimisation */
static void tree_ssa_gvn(void);

/* adds a block_expr to the first partition, in which a block contains
   all expressions of a certain type */
static void add_to_inital_partition(block_expr*);

/* adds block_expr be to the block b */
static void add_to_block(block** b, block_expr** be);

/* adds block b to partition p */
static void add_to_partition(block** p, block** b);

/* allocates a new block_expr with defines, expr, stmt and bb as given */
static block_expr* create_block_expr(tree defines, tree expr, tree stmt,
                                     basic_block bb);

/* creates a near perfect clone of be; the next pointer is set to 0 */
static block_expr* clone(block_expr* be);

/* allocated a new, empty block */
static block* create_block(void);

/* free()s everyting in partition p; sets p to 0 */
static void free_partition(block** p);

/* returns the block_expr in partition p that defines operand op */
static block_expr* defines(const block* p, const tree op);

/* returns the block in partition p that holds the block_expr be */
static block* in_block(block* p, const block_expr* be);

/* replaces reduntant statements and expressions, based on the
   partition calculated */
static void dominator_based_gvn(void);

/* replaces all uses of operand old by uses of operand replacement:
     x_5 = a_8 + 4;
     replace_uses(a_8, x_6);
     x_5 = x_6 + 4;
   also removes the statement stmt
*/
static void replace_uses(tree old, tree replacement, tree stmt);

static void dump(const char*, ...);

static void dump_gvn_stmt(tree);
static void dump_gvn_expr(tree);

/* prints the partition p on DUMPFILE, preceded by the text msg */
#ifdef DUMPFILE
static void dump_partition(block* p, const char* msg);
```

```
#endif

static void dump(const char* fmt, ...) {
#ifdef DUMPFILE
  va_list arg;

  va_start(arg, fmt);
  vfprintf(fd, fmt, arg);
  va_end(arg);
#else
  (void) fmt;
#endif
}

static void dump_gvn_stmt(tree s) { /* the name dump_stmt is already taken */
#ifdef DUMPFILE
  print_generic_stmt(fd, s, 0);
#else
  (void) s;
#endif
}

static void dump_gvn_expr(tree e) {
#ifdef DUMPFILE
  print_generic_expr(fd, e, 0);
#else
  (void) e;
#endif
}


/* function definitions */
static void
initialize(void)
{
#ifdef DUMPFILE
  static int first = 1;
  if (first) {
    unlink(DUMPFILE);
    first = 0;
  }
  if (!(fd = fopen(DUMPFILE, "a")))
    {
      fprintf(stderr, "tree-ssa-gvn.c: failed to open %s\n", DUMPFILE);
      exit(1);
    }
#endif
}

static void
finalize(void)
{
#ifdef DUMPFILE
  basic_block bb;
```

```
  dump_partition(gvn_partition,
                 "finalize() entered: the partition is as follows");
  dump_partition(gvn_const_partition,
                 "the partition containing constants and undefs:");

  /* dump what gvn returns to gcc */
  dump("o> --------- tree-ssa-gvn, exiting ---------\n");
  FOR_EACH_BB (bb)
    {
      tree phi;
      block_stmt_iterator iter;
      tree stmt;

      dump("o> visiting basic block %d\n", bb->index);
      for (phi = phi_nodes(bb); phi; phi = TREE_CHAIN(phi)) {
        dump("o> ");
        dump_gvn_stmt(phi);
      }
      for (iter = bsi_start(bb); !bsi_end_p(iter); bsi_next(&iter))
        {
          stmt = bsi_stmt(iter);
          dump("o> ");
          dump_gvn_stmt(stmt);
        }
    }
  /* end of dump */

  fclose(fd);
  fd = 0;
#endif

#ifdef VERBOSETIME
  fprintf(stdout, "GVN (cumulative):\tpart: %f ms\tgvn: %f ms\n",
          part_tot, gvn_tot);
#endif

#ifdef VERBOSEREPLACE
  fprintf(stdout, "GVN (cumulative):\tmod : %d stmts \trem: %d stmts\n",
          stmts_modified, stmts_removed);
#endif

  free_partition(&gvn_partition);
  free_partition(&gvn_const_partition);
}

static void
tree_ssa_gvn(void)
{
  basic_block bb;
#ifdef VERBOSETIME
  struct timeval start;
  struct timeval finish;
#endif
```

```
  initialize();
  dump("i> --------- tree-ssa-gvn, entering ---------\n");
  FOR_EACH_BB (bb)
    {
      tree phi;
      block_stmt_iterator iter;

      dump("i> visiting basic block %d\n", bb->index);
      for (phi = phi_nodes(bb); phi; phi = TREE_CHAIN(phi)) {
        dump("i> ");
        dump_gvn_stmt(phi);
        add_to_inital_partition(create_block_expr(PHI_RESULT(phi), phi, 0,bb));
      }

      for (iter = bsi_start(bb); !bsi_end_p(iter); bsi_next(&iter))
        {
          tree stmt;
          block_expr* be;


          stmt = bsi_stmt(iter);
          dump("i> ");
          dump_gvn_stmt(stmt);

          /* note that TREE_OPERAND(stmt, 0) of a 'return a + b' is
              '<retval> = a + b': see tree.def lines 824--829 */
          if (TREE_CODE(stmt) == RETURN_EXPR && TREE_OPERAND(stmt, 0))
            stmt = TREE_OPERAND(stmt, 0);


          if (TREE_CODE(stmt) == MODIFY_EXPR &&
              IS_INITAL_PARTITION_TYPE(TREE_CODE(TREE_OPERAND(stmt, 1))))
            {
              be = create_block_expr(TREE_OPERAND(stmt, 0),
                                     TREE_OPERAND(stmt, 1),
                                     stmt, bb);
              add_to_inital_partition(be);
            }
        }
    }

#ifdef VERBOSETIME
  gettimeofday(&start, 0);
#endif
  partition_n_square();
#ifdef VERBOSETIME
  gettimeofday(&finish, 0);
  part_tot += DIFF_IN_MSEC(start, finish);

  gettimeofday(&start, 0);
#endif
  dominator_based_gvn();
#ifdef VERBOSETIME
```

```
  gettimeofday(&finish, 0);
  gvn_tot += DIFF_IN_MSEC(start, finish);
#endif

  finalize();
}

static int
match(tree v, tree w, block* p)
{
  int i, max;

  if (TREE_CODE(v) == PHI_NODE)
    {
#ifdef VERBOSEMATCH
      dump("\nmatch(): about to compare these two PHI nodes:\n\t");
      dump_gvn_stmt(v);
      dump("\t");
      dump_gvn_stmt(w);
#endif
      if (PHI_NUM_ARGS(v) != PHI_NUM_ARGS(w))
        return FALSE;

      for (i = 0; i < PHI_NUM_ARGS(v); i++)
        {
          dump("\t\texamining arg(%d)\n\t\t", i);
          dump_gvn_stmt(PHI_ARG_DEF(v, i));
          dump("\t\t");
          dump_gvn_stmt(PHI_ARG_DEF(w, i));

          if (in_block(p, defines(p, PHI_ARG_DEF(v, i))) !=
              in_block(p, defines(p, PHI_ARG_DEF(w, i))))
            return FALSE;
        }
      return TRUE;
    }

#ifdef VERBOSEMATCH
  dump("\nmatch(): about to compare these two TREE nodes:\n\t");
  dump_gvn_stmt(v);
  dump("\t");
  dump_gvn_stmt(w);
#endif

  assert(TREE_CODE(v) == TREE_CODE(w));
  switch (TREE_CODE_CLASS(TREE_CODE(v)))
    {
    case 'c':
      dump("match(): comparing constants!\n");
      return TREE_INT_CST_LOW(v) == TREE_INT_CST_LOW(w);
    case '1': max = 1; break;
    case '2': max = 2; break;
    default: /* which should never occur */
      dump("match(): v is neither a unary nor a binary operation\n");
```

```
        abort();
        return FALSE;
      }

    for (i = 0; i < max; i++)
      if (in_block(p, defines(p, TREE_OPERAND(v, i))) !=
          in_block(p, defines(p, TREE_OPERAND(w, i))))
        return FALSE;
    return TRUE;
}

static void
partition_n_square(void)
{
    block *pi0, *pi1, *b, *same, *other;
    block_expr *v, *w, *c;
    int change;

    pi1 = gvn_partition;
    change = TRUE;

#ifdef VERBOSEPARTITION
    dump_partition(gvn_partition, "inital partition is as follows:\n");
#endif

    while (change)
      {
#ifdef VERBOSEPARTITION
        dump("partition_n_square(): entering while(change)\n");
#endif
        change = FALSE;
        pi0 = pi1;
        pi1 = 0;

        b = pi0;
        while (b)
          {
            other = 0;
            assert(b->expr);
            v = b->expr;
            c = clone(v);
            assert(c != v);
            assert(c->expr == v->expr);
            same = create_block();
            add_to_partition(&pi1, &same);
            assert(pi1 != 0);
            add_to_block(&same, &c);
            w = v->next;
            while (w)
              {
                c = clone(w);
                if (match(v->expr, w->expr, pi0))
                  {
                    add_to_block(&same, &c);
```

```
            }
          else
            {
              if (!other)
                {
                  other = create_block();
                  add_to_partition(&pi1, &other);
                  change = TRUE;
                }
              add_to_block(&other, &c);
            }
          w = w->next;
        }
      b = b->next;
    }

  free_partition(&pi0);
#ifdef VERBOSEPARTITION
  dump_partition(pi1, "partition_n_square(): loop end reached, "
                 "the partition is as follows");
#endif
}

  gvn_partition = pi1;
#ifdef VERBOSEPARTITION
  dump("partitioning complete!\n");
#endif
}

static block_expr*
create_block_expr(tree defines, tree expr, tree stmt, basic_block bb)
{
  block_expr* be;

  be = (block_expr*) xmalloc(sizeof(block_expr));
  be->next = 0;
  be->defines = defines;
  be->expr = expr;
  be->stmt = stmt;
  be->removed = FALSE;
  be->bb = bb;
  return be;
}

/* clones defines, expr and stmt, but sets next to 0 */
static block_expr*
clone(block_expr* be)
{
  block_expr* clone;

  clone = (block_expr*) xmalloc(sizeof(block_expr));
  clone->next = 0;
  clone->defines = be->defines;
  clone->expr = be->expr;
```

```
  clone->stmt = be->stmt;
  clone->removed = FALSE;
  clone->bb = be->bb;
  return clone;
}

static block*
create_block(void)
{
  block* b;

  b = (block*) xmalloc(sizeof(block));
  b->next = 0;
  b->expr = 0;
  return b;
}

static void
add_to_inital_partition(block_expr* be)
{
  block* b;

  b = gvn_partition;
  while (b)
    {
      if (TREE_CODE(b->expr->expr) == TREE_CODE(be->expr) &&
          (TREE_CODE(be->expr) != PHI_NODE ||
           bb_for_stmt(be->expr) == bb_for_stmt(b->expr->expr)))
        /* since phi(stmt) == 0, we use expr, which happens to be
           equal to stmt for phis. this is, admittedly, quite
           disgusting. refactoring of the block_expr is indeed
           motivated */
        {
          add_to_block(&b, &be);
          return;
        }
      b = b->next;
    }
  dump("creating new partition block for the [%s] type\n",
       tree_code_name[(int) TREE_CODE(be->expr)]);
  b = create_block();
  add_to_partition(&gvn_partition, &b);
  add_to_block(&b, &be);
}

static void
add_to_block(block** b, block_expr** be)
{
  (*be)->next = (*b)->expr;
  (*b)->expr = (*be);
}

static void
add_to_partition(block** p, block** b)
```

```
{
  (*b)->next = (*p);
  (*p) = (*b);
}

static void
free_partition(block** p)
{
  block *b, *bt;
  block_expr *be, *bet;

  b = (*p);
  while (b)
    {
      be = b->expr;
      while (be)
        {
          bet = be->next;
          free(be);
          be = bet;
        }
      bt = b->next;
      free(b);
      b = bt;
    }
  (*p) = 0;
}

/* returns the block_expr in partition partition that defines the
   operand operand */
static block_expr*
defines(const block* partition, const tree operand)
{
  block_expr* be;
  block* b;

  while (partition)
    {
      be = partition->expr;
      while (be)
        {
          if (be->defines == operand)
            return be;
          be = be->next;
        }
      partition = partition->next;
    }

  /* search const */
  b = gvn_const_partition;
  while (b)
    {
      int same, both_cst;
      tree def;
```

```
      assert(b->expr->next == 0);
      def = b->expr->defines;
      same = FALSE;
      both_cst = (TREE_CODE(operand) == INTEGER_CST &&
                  TREE_CODE(b->expr->defines) == INTEGER_CST);
      if (both_cst) {
        int a, b;
        a = TREE_INT_CST_LOW(operand);
        b = TREE_INT_CST_LOW(def);
        same = (a == b);

      }
      else if (TREE_CODE(operand) != INTEGER_CST &&
               TREE_CODE(b->expr->defines) != INTEGER_CST)
        same = (operand == b->expr->defines);
      if (same)
        {
#ifdef VERBOSEMATCH
          dump("int / pp:\t%d / ", (int) operand);
          dump_gvn_stmt(operand);
          dump("int / pp:\t%d / ", (int) b->expr->defines);
          dump_gvn_stmt(b->expr->defines);
#endif
          return b->expr;
        }
      b = b->next;
    }

  /* insert new block in const (only called during first iteration of
     partition_n_square: this is correct) */
  be = create_block_expr(operand, 0, operand, 0);
  b = create_block();
  add_to_partition(&gvn_const_partition, &b);
  add_to_block(&b, &be);
#ifdef VERBOSEMATCH
  dump("warning: defines() inserts new expr (%d)\n", (int) be);
#endif
  return be;
}

/* returns the block in partition partition in which block_expr be is */
static block*
in_block(block* partition, const block_expr* be)
{
  block_expr* bei;
  block* b;

  while (partition)
    {
      bei = partition->expr;
      while (bei)
        {
          if (bei == be)
```

```
            return partition;
          bei = bei->next;
        }
      partition = partition->next;
    }

#ifdef VERBOSEMATCH
  dump("warning: in_block() fall-through for be->defines = ");
  dump_gvn_stmt(be->defines);
#endif
  b = gvn_const_partition;
  while (b)
    {
      assert(b->expr);
      if (b->expr == be)
        {
#ifdef VERBOSEMATCH
          dump("\t(%d) be->defines == ", (int) be);
          dump_gvn_stmt(be->defines);
          dump("\t(%d) b->expr->defines == ", (int) b->expr);
          dump_gvn_stmt(b->expr->defines);
#endif
          return b;
        }
      b = b->next;
    }
  /* by now we should have found the block, so aborting: */
  abort();
  return 0;
}

#ifdef DUMPFILE
static void
dump_partition(block* p, const char* header)
{
  block_expr* expr;

  dump(header);
  dump("\n--------------------------------\n");
  while (p)
    {
      int brackets;

      expr = p->expr;
      brackets = (int) expr->expr;
      if (brackets) /* fails for the consts and undefs blocks */
        dump("\tpartition block: %s {\n",
             tree_code_name[(int) TREE_CODE(expr->expr)]);
      while (expr)
        {
          dump("\t\t");
          dump_gvn_stmt(expr->stmt ? expr->stmt : expr->expr);
          expr = expr->next;
        }
```

```
      if (brackets)
        dump("\t}\n");
      p = p->next;
    }
  dump("--------------------------------\n");
}
#endif

static void
dominator_based_gvn(void)
{
  block* b;
  block_expr* v, *w;

  for (b = gvn_partition; b; b = b->next)
    {
      if (!b->expr->next) /* don't bother if only one element in the block */
        continue;

      for (v = b->expr; v; v = v->next)
        for (w = b->expr; w; w = w->next) {
          /* note that, per definition, a basic block bb always
             dominates itself */
          if (v != w && !v->removed && !w->removed)
            {
              dump("considering possible pair: ");
              dump_gvn_stmt(v->stmt ? v->stmt : v->expr);
              dump(" and: ");
              dump_gvn_stmt(w->stmt ? w->stmt : w->expr);
              dump(" bbs: %d, %d\n",
                        v->bb->index, w->bb->index);
              dump(" w->bb dom b->bb: %d\n",
                        dominated_by_p(CDI_DOMINATORS, w->bb, v->bb));
            }

          if (v != w && !v->removed && !w->removed &&
              dominated_by_p(CDI_DOMINATORS, w->bb, v->bb))
            {
              vuse_optype vuses;
              vdef_optype vdefs;
              def_optype defs;
              use_optype uses;
              stmt_ann_t ann;
              size_t i;
              tree stmt;

              if (!v->stmt) /* phi node */
                {
                  dump("found a pair: ");
                  dump_gvn_stmt(v->expr);
                  dump(" and: ");
                  dump_gvn_stmt(w->expr);
                  replace_uses(w->defines, v->defines, w->expr);
                  w->removed = TRUE;
```

```
        }
      else /* anything but a phi node */
        {
          dump("found a pair: ");
          dump_gvn_stmt(v->stmt);
          dump(" and: ");
          dump_gvn_stmt(w->stmt);

          stmt = v->stmt;
          dump("(int) stmt == %d\n", (int) stmt);
          dump("now printing information of uses etc for stmt ");
          dump_gvn_stmt(stmt);
          get_stmt_operands(stmt);
          ann = stmt_ann(stmt);

          defs = DEF_OPS(ann);
          for (i = 0; i < NUM_DEFS(defs); i++)
            {
              dump("\t DEF: ");
              dump_gvn_expr(DEF_OP(defs, i));
              dump("\n");
            }

          uses = USE_OPS(ann);
          for (i = 0; i < NUM_USES(uses); i++)
            {
              dump("\t USE: ");
              dump_gvn_expr(USE_OP(uses, i));
              dump("\n");
            }

          vdefs = VDEF_OPS(ann);
          for (i = 0; i < NUM_VDEFS(vdefs); i++)
            {
              dump("\tVDEF: ");
              dump_gvn_expr(VDEF_OP(vdefs, i));
              dump("\n");
            }

          vuses = VUSE_OPS(ann);
          for (i = 0; i < NUM_VUSES(vuses); i++)
            {
              dump("\tVUSE: ");
              dump_gvn_expr(VUSE_OP(vuses, i));
              dump("\n");
            }

          dump("now done printing\n");
          replace_uses(TREE_OPERAND(w->stmt, 0),
                       TREE_OPERAND(v->stmt, 0), w->stmt);
          w->removed = TRUE;
        }
    }
}
```

```
    }
}

static void
replace_uses(tree a, tree b, tree remove)
{
  basic_block bb;

  dump("this is replace_uses(), replacing uses of ");
  dump_gvn_expr(a);
  dump(" by uses of ");
  dump_gvn_expr(b);
  dump(", and I will remove the statement ");
  dump_gvn_stmt(remove);

  FOR_EACH_BB (bb)
    {
      tree phi, prev;
      block_stmt_iterator iter;
      tree stmt;
      stmt_ann_t ann;
      size_t i;
      use_optype uses;
      vuse_optype vuses;

      /* scan phi nodes */
      prev = NULL_TREE;
      for (phi = phi_nodes(bb); phi; phi = TREE_CHAIN(phi)) {
        int i;

        if (phi == remove)
          {
            dump("found the phi to remove [%d]: ", (int) phi);
            dump_gvn_stmt(phi);
#ifdef VERBOSEREPLACE
            ++stmts_removed;
            fprintf(stdout, "GVN removed ");
            print_generic_stmt(stdout, phi, 0);
#endif
            remove_phi_node(phi, prev, bb);
            continue;
          }
        for (i = 0; i < PHI_NUM_ARGS(phi); i++)
          {
            if (PHI_ARG_DEF(phi, i) == a)
              {
#ifdef VERBOSEREPLACE
                ++stmts_modified;
#endif
                PHI_ARG_DEF(phi, i) = b;
                modify_stmt(phi);
              }
          }
        prev = phi;
```

```
    }

    /* on to ordinary stmts */
    for (iter = bsi_start(bb); !bsi_end_p(iter); )
      {
        stmt = bsi_stmt(iter);
        if (stmt == remove)
          {
            dump("replace> found the stmt to remove: ");
            dump_gvn_stmt(stmt);
#ifdef VERBOSEREPLACE
            ++stmts_removed;
            fprintf(stdout, "GVN removed ");
            print_generic_stmt(stdout, stmt, 0);
#endif
            bsi_remove(&iter);
            dump("last stmt == %d\n", bsi_end_p(iter));
            if (bsi_end_p(iter)) /* if this was the last stmt... */
              continue; /* ...then break the loop... */
            stmt = bsi_stmt(iter); /* ...otherwise, drop through */
          }

        dump("replace> examining: ");
        dump_gvn_stmt(stmt);

        get_stmt_operands(stmt);
        ann = stmt_ann(stmt);
        uses = USE_OPS(ann);
        vuses = VUSE_OPS(ann);

        for (i = 0; i < NUM_USES(uses); i++)
          {
            if (USE_OP(uses, i) == a)
              {
#ifdef VERBOSEREPLACE
                ++stmts_modified;
#endif
                USE_OP(uses, i) = b;
                modify_stmt(stmt);
                dump("\tstmt on previous line was [MODIFIED] (use)\n");
              }
          }
        for (i = 0; i < NUM_VUSES(vuses); i++)
          {
            if (VUSE_OP(vuses, i) == a)
              {
#ifdef VERBOSEREPLACE
                ++stmts_modified;
#endif
                VUSE_OP(vuses, i) = b;
                modify_stmt(stmt);
                dump("\tstmt on previous line was [MODIFIED] (vuse)\n");
              }
          }
```

```
            bsi_next(&iter);
          }
      }
}

static bool
gate_gvn (void)
{
  return flag_tree_gvn != 0;
}

/* see tree-pass.h for documentation on PROP_* etc */
struct tree_opt_pass pass_gvn =
{
    "gvn", /* name of optimisation pass */
    gate_gvn, /* exec gvn iff this func returns non-null */
    tree_ssa_gvn, /* entry point */
    NULL, /* not used */
    NULL, /* not used */
    0, /* static pass number */
    TV_TREE_GVN, /* timevar id */

    PROP_cfg | PROP_ssa, /* properties required */
    0, /* properties provided */
    0, /* properties destroyed */

    TODO_verify_all, /* things to do before running gvn */
    TODO_verify_all /* things to do after running gvn */
};
```

Makefile.in:870	tree−ssa−gvn.o
timevar.def:76	DEFTIMEVAR (TV_TREE_GVN, "tree_GVN")
tree-optimize.c:319	NEXT_PASS (pass_gvn);
tree-pass.h:107	**extern struct** tree_opt_pass pass_gvn;

Table A.1: Modifications related to GVN, excluding the `tree-ssa-gvn.c` file

common.opt:739–741	ftree −gvn Common Enable SSA−GVN optimisation on trees
flags.h:736	**extern int** flag_tree_gvn;
opts.c:1499–1501	**case** OPT_ftree_gvn: flag_tree_gvn = value; **break**;
toplev.c:829	**int** flag_tree_gvn = 0;

Table A.2: Modifications related to command line option framework

Other GVN related files

See Table A.1 for a listing of the files that needed modification in order to incorporate GVN in GCC.

A.2.2 The command line option framework

See Table A.2 for the additions made to enable the `-f[no-]gvn` options.

Bibliography

[1] A. W. Appel, *Modern compiler implementation in Java*, Cambridge University Press, rev and expanded ed., 1999.
Work detailing how to build an optimising compiler from scratch, working its way from the parser to the support of object oriented languages and garbage collection.

[2] P. Cederqvist et al., *Version Management with CVS*, 2003.
The official manual for the Concurrent Version System, a tool for versioning and sharing source code, commonly used by companies and open source project: specifically, used in the gcc project. Available for download at http://www.cvshome.org/docs/manual/.

[3] C. Click, *Global code motion/global value numbering*, in Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, ACM Press, 1995, pp. 246–257.
Alternative implementation of gvn, presented along with studies of its interactions with serveral other optimisations.

[4] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman, *Efficiently computing static single assignment form and the control dependence graph*, ACM Trans. Prog. Lang. Syst., (1991).
Paper on how to efficiently convert a program to ssa form by the introduction of dominance frontiers.

[5] *The GNU general public license*.
Document describing the terms of usage and copying of software under the GNU General Public License. Shipped together with software released under said license; available online at http://www.gnu.org/licenses/gpl.txt.

[6] *The GNU homepage*. Located at http://www.gnu.org.
The GNU project website, with links to its many subprojects.

[7] D. E. Knuth, *The TEXbook*, Addison-Wesley, 1984.
The definitive manual on the TEX system, written by the inventor himself.

[8] D. Novillo, *Design and implementation of Tree SSA*, in Proceedings of the gcc Developers' Summit, 2004, pp. 119–130.
Introduction to the implementation of ssa in gcc, written by the maintainer of that cvs development branch and presented at the gcc Summit 2004. Available online at http://www.gccsummit.org/.

[9] 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, ACM Press, 1988, pp. 12–27.
Paper on early experiments with value numbering, ssa form and ϕ-functions.

[10] L. T. Simpson, *Value-Driven Redundancy Elimination*, PhD thesis, Rice University, April 1996.
Thesis which includes, among other things, detailed discussions on how to implement value numbering efficiently.

[11] J. Skeppstedt, *An introduction to the theory of optimizing compilers with performance measurements*. Work in progress, distributed locally at Lund Institute of Technology, Lund University, 2002.
Work presenting plenty of compiler optimisations in scrutinising detail.

[12] R. M. Stallman, *GNU Compiler Collection Internals*.
Document accompanying the gcc sources; a manual detailing design and overall layout of the gcc code.

[13] M. N. Wegman and F. K. Zadeck, *Constant propagation with conditional branches*, ACM Trans. Prog. Lang. Syst., (1991).
Discussion on how several previous implementations of Constant Propagation was merged into the eponymous version