

Certificate of Approval

This is to certify that the project, **Optimizing GCC for Intel Pentium IV Architecture**, undertaken by **Vijayendra Bose, Aditya Joshi, Nilesh Tayade, and Kedar Thakur**, has been satisfactorily completed as per the requirements of the term work of the project for the degree of **Bachelor of Engineering** as prescribed by the University of Mumbai.

Guides

1. Internal _____

2. External _____

Examiners

1. Internal _____

2. External _____

Head of the Department

Principal

College Seal

Date:

Acknowledgments

First of all, we thank our guide, **Prof. S. Biswas**, CSE, IITB, for his encouragement, support and advice throughout this project.

We also thank, **Prof. Sukhwant K.**, Fr. C.R.I.T., our internal guide during this project.

From IITB, we thank:

Mr. Nitin Jain, for his initial work on GCC, that served as a reference throughout our project.

Mr. Rajesh Babu S., for his work on the IITB patch for Abacus and research on the HSTF algorithm.

Mr. Naresh Bhakar, for helping us with GCC installation, configuration and other issues throughout the project.

From Fr. C.R.I.T., we thank:

Dr. H. K. Kaura, Head of Computer Engineering Department, for guiding us and reviewing our project posters.

Prof. Amroz Siddiqui and **Prof. Marvin Mendonca** for providing us access to their computing facilities, which helped us greatly during this project.

For software contributions, we thank:

phpBB Group and **phpMyAdmin Team**, for the style sheets used in the GCC Code Analysis Tool.

Agnel GCC Group
11th April 2006

Abstract

One of the long term and large scale research investigations at IIT Bombay centers around GCC (GNU Compiler Collection). This project is a part of these investigations. The main aim of this project was to improve the performance of GCC on Intel Pentium IV processors. Firstly, we developed a patch for predicting register usage for expressions involving basic arithmetic operators. Secondly, we found four optimizations in the assembly code currently being produced by GCC. Thirdly, we developed a GCC study tool in PHP, that contains several utilities useful in code analysis, as well as implementations of the optimizations. And lastly, we investigated ways of incorporating the HSTF algorithm into GCC.

List of Abbreviations

ADDL	Addition
ANSI	American National Standards Institute
AS	(GNU) ASsembler
ASIP	Application Specific Instruction-set Processor
AST	Abstract Syntax Tree
CISC	Complex Instruction Set Computer
CPP	(GNU) C Pre-Processor
CRT	C RunTime libraries
CSE	Common Subexpression Elimination
DAG	Directed Acyclic Graph
DCE	Dead Code Elimination
DRDO	Defence Research & Development Organisation
DSP	Digital Signal Processor
EAX	Extended Accumulator Register
ECX	Extended Counter Register
FORTRAN	FORmula TRANslator
GCC	GNU Compiler Collection / GNU C Compiler
GCJ	GNU Compiler for Java
GCL	GNU Common Lisp
GST	GNU SmallTalk
GNU	GNU is Not Unix
GPL	General Public License
HSTF	Heavier SubTree First
ICC	Intel C++ Compiler
IITB	Indian Institute of Technology Bombay
LEAL	Load Effective Address
LSTF	Left SubTree First
md	Machine Description
MMX	Multi Media eXtensions
MOVL	Move
NEGL	Negation
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Language
RTX	RTL Expressions
SIMD	Single Instruction Multiple Data
SPARC	Scalable Processor ARChitecture
SSE	Streaming SIMD Extensions
SUBL	Subtraction

List of Figures

1.1 Timeline Chart	13-14
2.1: A sample expression tree	17
2.2: Order of expression evaluation in LSTF	18
2.3: Control Flow in LSTF algorithm	19
3.1: Order of expression evaluation in HSTF	22
3.2: Control Flow in HSTF algorithm	23
3.3: Level 0 DFD of the project.	29
3.4: Level 1 DFD of the project.	29
3.5: Level 2 DFD (Part 1) of the project.	30
3.6: Level 2 DFD (Part 2) of the project.	30
5.1: Register Usage Report	40
5.2: Main Control Panel	41
5.3: Optimization Input Screen	42
5.4: Optimization Result	42
5.5: GCC basics FAQ	43

Contents

Certificate of Approval	1
Acknowledgments	2
Abstract	3
List of Abbreviations	4
List of Figures and Tables	5
1. Introduction	8
1.1 Aim and Motivation	8
1.2 Compilers	8
1.3 GCC	8
1.4 Project Scope	10
1.5 Software Model	10
1.6 Team Organization	12
1.7 Timeline Chart	12
2. System Analysis	15
2.1 Existing System	15
2.2 GCC on Intel Pentium IV	15
2.3 The Generated Assembly Code	15
2.4 The LSTF Algorithm	16
2.5 Software and Hardware Requirements	20
3. Proposed System and Design	21
3.1 System Components	21
3.2 Improved Instruction Selection	21
3.3 The HSTF Algorithm	21
3.4 The IITB patch	23
3.5 The GCC Code Analysis Tool	27
3.6 Register Usage Prediction	28
3.7 Data Flow Diagrams	29
4. System Implementation	31
4.1 Problems in HSTF implementation	31
4.2 The Prediction Algorithms	32
4.3 The Agnel GCC Group Patch	33

4.4 Assembly Optimizations	35
4.5 GCC Tool features	37
5. Performance Measurement	38
5.1 Prediction Accuracy	38
5.2 Optimization Timings	39
5.3 GCC Tool screen shots	40
6. Conclusions & Future Work	44
6.1 Conclusion	44
6.2 A Better Metric	44
6.3 Improved Instruction Selection	44
6.4 Extending the GCW Patch	45
6.5 Machine Description for Pentium IV	45
Appendices	46
A.1 GCC Installation	46
A.2 Patching Basics	48
References	50
Bibliography	51

1. Introduction

1.1 Aim and Motivation

One of the long term and large scale research investigations at IIT Bombay centers around GCC (GNU Compiler Collection). This project is a part of these investigations. The main aim of this project is to improve the GCC performance on Intel Pentium IV processor. Towards this end, we performed a set of activities, which are discussed in the later sections. Before getting down to the specifics, we provide a brief overview of the area of operation, compilers and GCC.

1.2 Compilers

A compiler is a system program that converts the input source code to a target code. The target code maybe assembly code or machine language code. Most of the software written today is written in high level languages. Hence, the role of the compiler assumes significance. There are many different compilers available today. E.g.: the Turbo C compiler, javac, the java compiler, etc. GCC is one such compiler.

1.3 GCC

In this section, we discuss the historical background of GCC and the ongoing GCC project at IIT Bombay. See [GOUG] for more details.

An Introduction

- GCC stands for GNU Compiler Collection
- It is a retargetable compiler framework
- It supports a variety of source (input) languages & target machines
- It has the ability to add support for new languages and machines
- It is a free software and is available under the GNU GPL (General Public License)

A Brief History

- The GNU Project was started in 1984 to create a complete Unix-like operating system as free software, in order to promote freedom and cooperation among computer users and programmers.
 - GCC was developed from scratch by the GNU Project
 - GCC was first Released in 1987
 - Richard M. Stallman was the original author of GCC.
-
- It was the first portable ANSI C optimizing compiler that was a free software
 - Since then, GCC has been an important tool in the free software movement

The GCC Project at IIT Bombay

- The GCC Project is one of the long-term and large-scale research investigations at IIT Bombay. See [JAIN] for more details.
- The long-term goals of this project are as follows:
 1. Automating the task of writing efficient machine descriptions
 2. Specification and Generation of Optimiser
 3. Automatic generation of the optimiser
 - 3.1 Testing of optimisations
 - 3.2 Compiled Code Verification
 4. Improving the Code Generator
 5. Improving machine descriptions
 6. Improving GCC
 7. Post pass transformations

The area that we are concerned with is Register Efficient Expression Evaluation.

Register Efficient Expression Evaluation

- Register Efficient Expression Evaluation strategy proposes the use of the HSTF (Heavier Sub Tree First) algorithm instead of the currently used Left Sub Tree First (LSTF) algorithm. This algorithm reduces the register requirements of any executing expression, thereby improving performance.

GCC Features

This section discusses the major features of GCC

Adaptable Compiler

- GCC runs on most available platforms today
- GCC can produce generated code for PCs, DSPs (including TMS series), 64-bit CPUs, micro-controllers, etc

Cross Compiler

- GCC is a native compiler as well as a cross compiler
- This means that it can produce generated code for a system that is different from the one on which GCC itself is running
- This allows it to produce code for systems not capable of running a compiler such as Embedded Systems

Self Compiler

- GCC is written in C, with a focus on portability
- It can compile itself and easily adapt to new systems

Multiple Language Front-ends

- GCC can compile and cross compile many languages for many different machine architectures
- This is possible due to the use of an intermediate language, RTL, as we shall see
- Examples include generation of C code for a supercomputer, ADA code for an embedded system

Modular Design

- The Modular design of GCC enables new language and machine support to be added easily
- GCC uses RTL, an intermediate language as an interface between the source language and the assembly language.

Free Software

- GCC is free software. It is free to use, modify, enhance, use other people's enhancements, and share your own enhancements
- This freedom has played an important role in the development of GCC

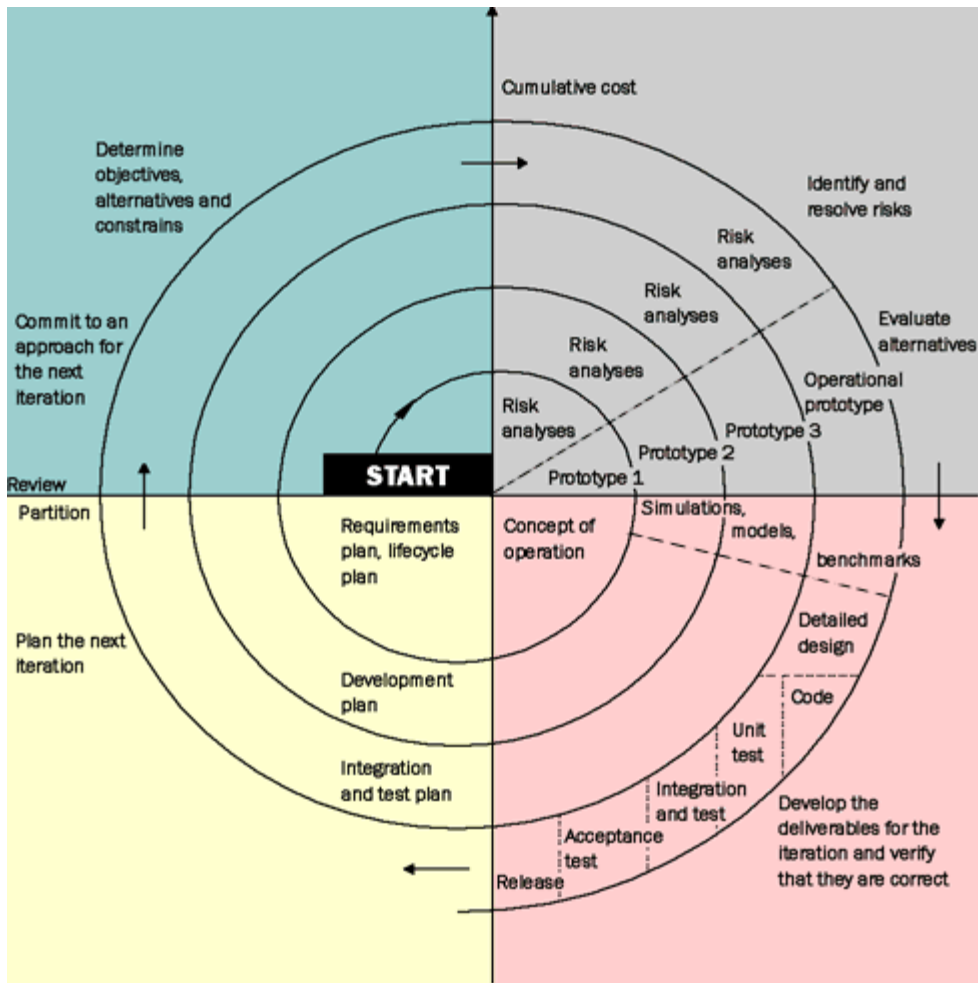
1.4 Project Scope

The scope of the project includes three key areas:

1. Investigation of the issues involved in the implementation of HSTF for GCC on Intel Pentium IV machines
2. The development of algorithms for predicting the register usage, and the number of stores performed by a given expression.
3. The search for assembly code optimizations
4. The development of a GCC code analysis tool.

1.5 Software Model

The "Spiral Model" of software development was used for this project. The **spiral model** is a software development process combining elements of both design and prototyping-in-stages, in an effort to combine advantages of top-down and bottom-up concepts.



A brief idea about the spiral nature of this project is as follows:

- We worked on providing the basic algorithms first.
- Then, in the next iteration, we implemented the algorithms
- We started from a bare bones approach added more functionalities as time permitted.
- This approach was also used in the development of the GCC code analysis tool.

1.6 Team Organization

The team organization for this project was “Democratic Decentralization”. The working of the team members in this project can be summarized as follows:

- None of the members assumed a central controlling role.
- Periodic goals were set describing the tasks to be completed
- Members divided the tasks amongst themselves.

- They regularly informed other members of the progress they had made in the task.

The role of the guides

- The guides were consulted at regular intervals and updated about the team's progress.
- They evaluated the team's progress and analyzed the research findings.
- The guides gave directions to the team members about the future course of action.

1.7 Time line Chart

This section describes the time-line chart for the project. A time-line chart is a chart, which describes the work done on a scale of time, throughout this project.

2. System Analysis

2.1 Existing System

The existing system for this project is GCC version 3.3.3 running on a GNU/Linux operating system on an Intel Pentium IV processor. We shall examine various aspects of this existing system and try to improve upon them.

2.2 GCC on Intel Pentium IV

The Pentium IV processor manufactured by Intel, is one of the widely used processors of this day. The performance of Pentium IV on GCC needs to be improved, because:

- There is no machine description (md) specifically for Pentium IV.
- Pentium IV instructions are not utilized properly, resulting in inefficient code.

Now, we consider, some aspects that need improvement.

2.3 The Generated Assembly Code

The first area that we look at is improving the assembly code produced by GCC.

Assembly Code

The GNU C Compiler converts the source code into an intermediate form (called as RTL). It then uses the machine description of the computer to find the assembly instructions available to execute the RTL code.

The assembly code generated by GCC may be examined using the `-S` switch. For example, to see the assembly code for the file `hello.c`, the command given is:

```
$ gcc -S hello.c
```

This will produce the assembly code file `hello.s`.

Quality of Assembly Code

Upon careful examination of the assembly code produced by GCC, we can conclude that it is not the most efficient code. There is always some scope for improvement. This is obvious since, an experienced assembly programmer can always do a better job.

Our goal is to find as many generalized improvements as possible by examining the assembly code.

Better instruction selection

Better instruction selection is one major class of improvements. On CISC machines like Pentium IV, there are a large number of instructions. Typically there are several ways of doing a given task.

For Example: The operation $a = b + c$ can be performed in two ways:

1. By using the complex LEAL instruction as:

leal (b,c) , a

2. By using the simple ADDL & MOVL instructions

addl c,b

movl b,a

One of these ways is better than the other. In this case, the second one is better than the first.

2.4 The LSTF Algorithm

The second area we look at is the use of LSTF algorithm. GCC uses the Left Subtree First or LSTF algorithm for expression tree traversal. This is the current state of the art. We shall now, discuss this algorithm in detail.

An Example

Consider an arithmetic expression: $(a+b) * ((c+d) * ((e+f) * (g+h)))$

The expression tree generated for this expression is as follows:

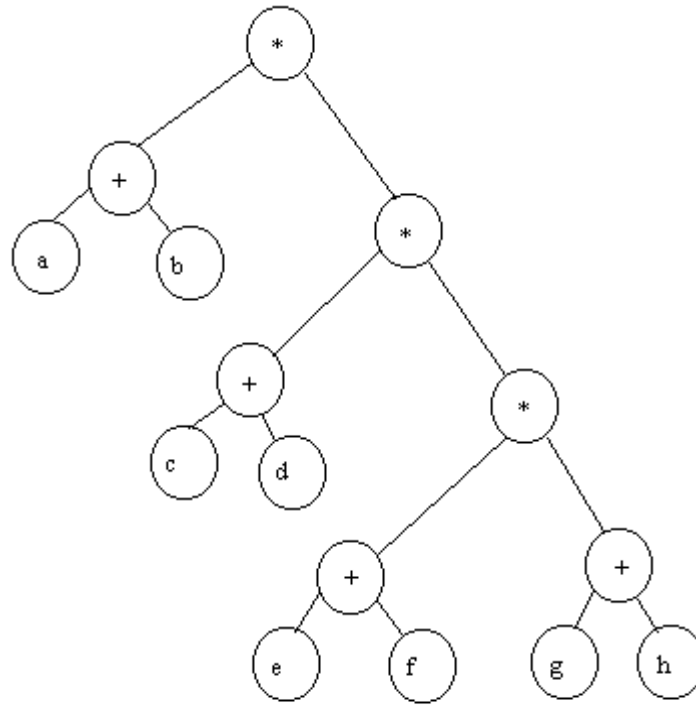


Figure 2.1: A sample expression tree

The LSTF approach

LSTF traverses the tree as follows:

First, it visits the left subtree of the root node, rooted at '+' which is "a+b". Subsequently, it traverses every left subtree in the order: "a+b", "c+d", and "e+f"

Merits of LSTF

1. Ease of implementation is a key advantage of LSTF. If you have to go left every time, there is no decision to be made. This saves some computational effort.
2. LSTF needs no storage for "heaviness" values as HSTF does. Hence, it needs less memory.

Demerits of LSTF

1. This strategy is very inflexible and does not take the tree structure into account
2. It performs poorly for right-heavy trees.

The following tree illustrates the order in which the expressions of the tree are evaluated as per LSTF algorithm:

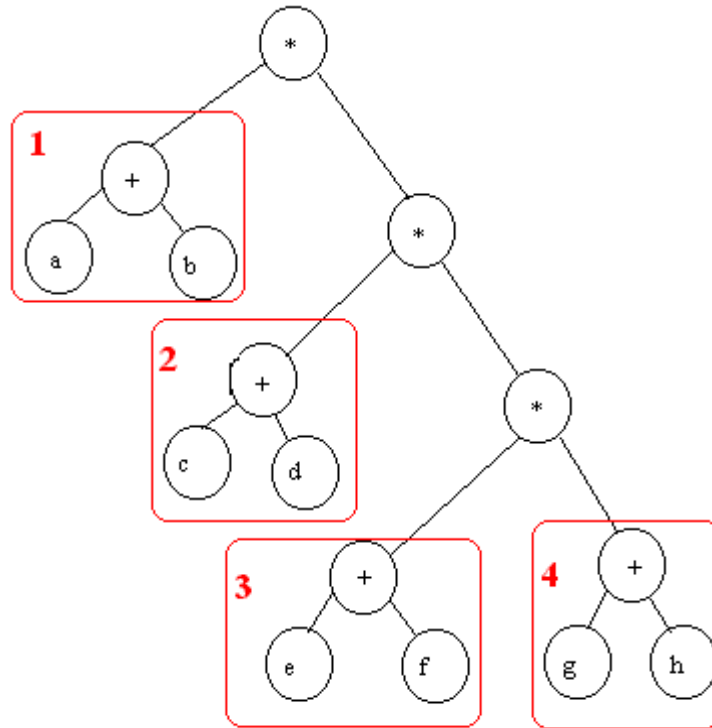


Figure 2.2: Order of expression evaluation in LSTF

Now, we define some terms we shall use in the subsequent discussion.

Decision Node

A decision node in an expression tree is a node with the following characteristics:

1. At least one of its children is an operator
2. At such a node, a decision needs to be taken about the direction in which the traversal should proceed.

E.g.: All multiplication nodes in the above figure are decision nodes.

The root of an expression tree is always the first decision node.

Control Flow

The sequence of expression tree nodes, a traversal algorithm visits during its execution is called as control flow.

LSTF Control Flow

The flow of control in LSTF traversal is illustrated as follows:

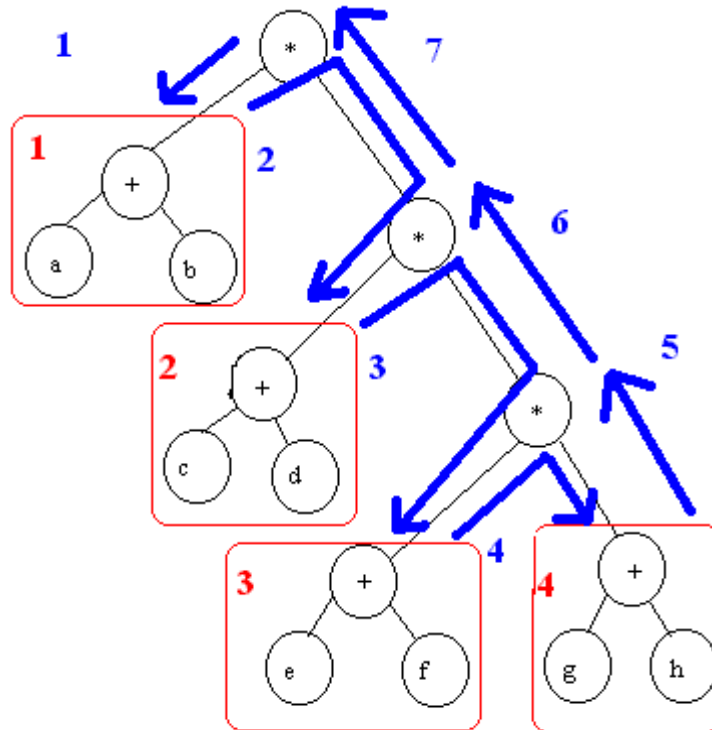


Figure 2.3: Control Flow in LSTF algorithm

Notes on LSTF

One thing to be noted is that LSTF used in GCC is not a trivial algorithm blindly opting for the left subtree at every decision node. It includes several tree optimizations. These optimizations change the tree structure and make it better suited to LSTF traversal.

An example of this is as follows: In the above figure, all of the decision nodes are multiplications. **Expression:** $(a+b) * ((c+d) * ((e+f) * (g+h)))$

If these are replaced by an operation such as addition or subtraction, then the order of operation execution is immaterial. **Expression:** $(a+b) + ((c+d) + ((e+f) + (g+h)))$

Here, LSTF balances the tree, so that performance penalty that would have resulted through the use of original tree-LSTF combination is reduced.

Assumption

The tree submitted for traversal is assumed to be pre-optimized. All the balancing, tree optimizations, etc. are performed prior to the LSTF/HSTF traversal. We deal only with tree traversal.

There are several shortcomings in the LSTF algorithm that are replaced by the Heavier Subtree First or the HSTF algorithm. We shall discuss HSTF in the next chapter.

2.5 Software and Hardware Requirements

The requirements for this project are as follows:

Hardware Requirements

Intel Pentium IV processor

Software Requirements

GNU/Linux Operating System, GCC 3.3.3 , Apache 2.0.0, PHP 5.0

3. Proposed System and Design

3.1 System Components

The system consists of three components, a register usage prediction algorithm, assembly optimization by better instruction selection, and a GCC code analysis tool. These components are described in detail in the next three sections of the report.

3.2 Improved Instruction Selection

To improve the instruction selection and the quality of assembly code in general, we take the following approach:

- Conduct research on the cycle times taken for the execution of various instructions.
- Run sample programs to find scopes of improvement in the assembly code produced by GCC.
- Conclusively prove that it is a valid improvement, by rigorous testing.
- Look at ways of implementing the improvement into GCC.

3.3 The HSTF Algorithm

The HSTF approach

HSTF traverses the tree as follows:

At every node, it opts for the heavier subtree. In this case, it opts for the right subtree at the root node and the next decision node.

Merits of HSTF

1. Takes tree structure into account during traversal process.
2. Performs equally well for all kinds of trees. (left-heavy, right-heavy or balanced)

Demerits of HSTF

1. Extra computations for heaviness at each node
2. Extra storage for heaviness values

The demerits of HSTF are more than compensated by its merits. Hence, HSTF is a **better** algorithm than LSTF.

The following tree illustrates the order in which the expressions of the tree are evaluated as per HSTF algorithm:

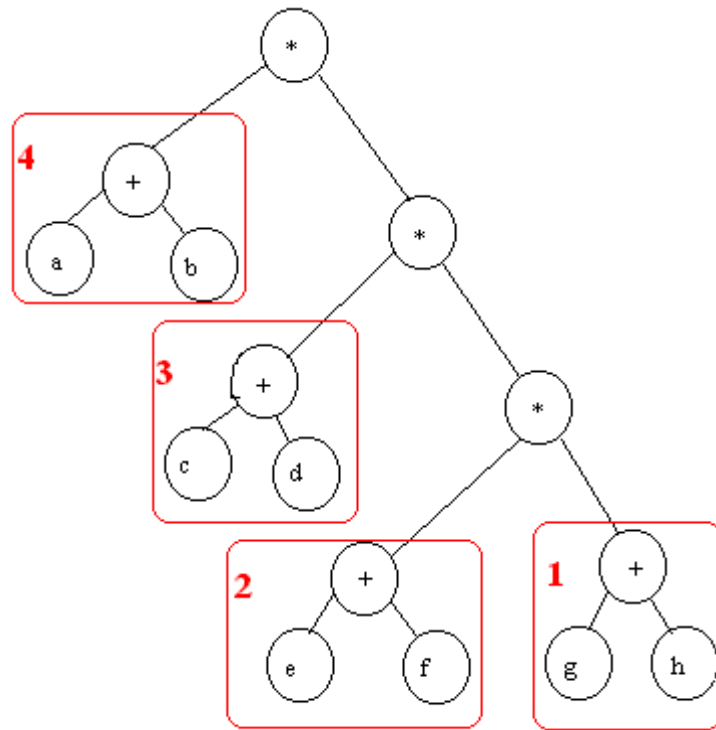


Figure 3.1: Order of expression evaluation in HSTF

Note: The execution of the 1st and the 2nd expressions maybe interchanged i.e., “e+f” before “g+h” or vice versa. This is because, the tree is balanced at that point, and any of the paths maybe chosen.

Control Flow

The expressions are executed in the above order. This is the result of decisions made at every decision node starting from the root node. This process is illustrated in the figure below:

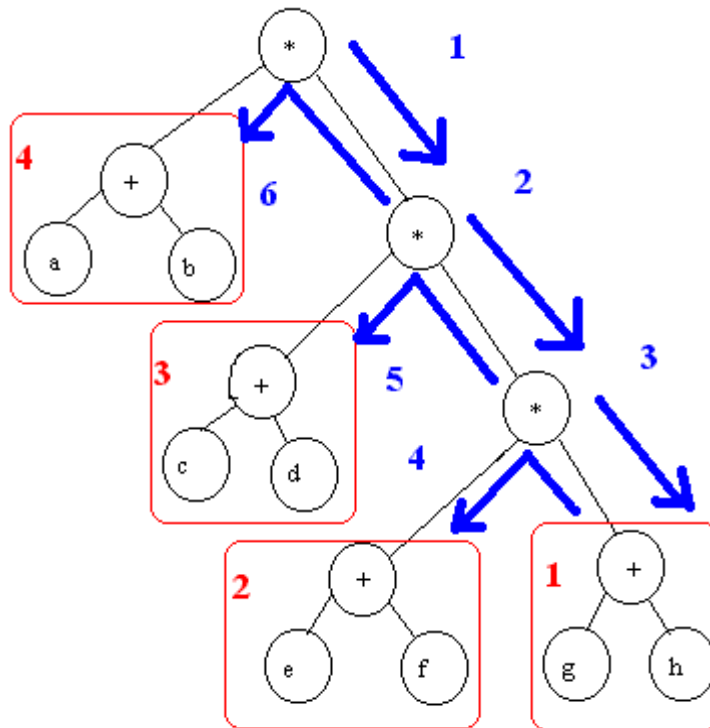


Figure 3.2: Control Flow in HSTF algorithm

3.4 The IITB patch

Introduction

1. HSTF is a known algorithm. It is based on the Sethi Ullman algorithm. It has proved theoretically better than LSTF.

2. GCC always evaluates any expression tree by LSTF. In doing so, it wastes registers & stack space (allocated for activation records).

Using LSTF also leads to extra instructions being generated in the assembly code. This wastes execution resources as well as CPU time.

All this can be avoided, if only, we switch to HSTF.

3. Benefits of HSTF are proportional to the expression size. Bigger the expression, better is the performance of HSTF.

Implementation (in Theory)

1. Now, you know that we should use the HSTF approach.

If we need to process the *heavier* subtree first, we should KNOW which of the subtrees is *heavier*.

What is heaviness ?

Heaviness of a tree node is the number of registers needed to evaluate the subtree rooted at that node. Heaviness can also be considered as the amount of resources used in evaluating the given subtree.

This is done by assigning weights to all the tree nodes.

So, every time you reach a non-leaf node, and are faced with the decision of whether to go right or left, all you have to do is to look at the weights of the nodes.

2. The next question is how do we assign these heaviness values to tree nodes ?

The existing IITB patch has been built for the Abacus architecture. Abacus is a RISC machine. Here, the operands of any instruction can be only immediate or register.

Thus, the assignment works as follows:

weight = 1 if register operand

weight = 0 if immediate operand

& weight(parent) = max(weight(children)) if children nodes have unequal weights

& weight(parent) = weight(child) + 1 if children nodes have equal weights.

3. We have an annotated tree. A tree where each node has a heaviness value. Now, all we have to do is to change the default traversal strategy used by GCC from LSTF to HSTF.

So, the theoretical implementation can be summarized as:

1. Assign weight (heaviness) values to tree nodes
2. Traverse the tree as per values assigned in (1)

Implementation (in Practice)

Let us see how do we implement the two theoretical tasks in reality.

Recall the two tasks:

1. Assign weight (heaviness) values to tree nodes
2. Traverse the tree as per values assigned in (1)

In GCC, the function `expand_expr()` traverses the tree and produces RTL. So, assigning the weights has to be done BEFORE `expand_expr` & HSTF traversal has to be incorporated WITHIN `expand_expr`.

So, the implementation strategy is:

```
....  
....  
....  
assign_weight() ; /* Call a function that will assign the weights */  
  
expand_expr();      /* Modified expand_expr that will traverse as per the weights*/  
....  
....  
....
```

1. A place to store the weight

Before we write a function to assign weights, we need a place to store these weight values.

GCC nodes are structures.

They currently, do not have any variable to keep track of node weight.

All we have to do is to add a field 'weight' (an int) to it.

So, all the GCC nodes will have a weight field. A value, which we can set and use as we go along.

This is the easiest part.

It is done by modifying the "tree_comon" structure in the file "tree.h" as shown below:

```
struct tree_common GTY(())  
{  
.....  
.....  
int weight;  
  
};
```

Only one line is added at the end as shown.

2. Assigning weights to nodes.

A function `assign_weight()` has been written for this purpose.

The function takes a pointer to a tree node as input & assigns heaviness values to it recursively.

What we do is, pass the root of the expression tree to `assign_weight()`. This does the job of assigning weights.

Hence, the implementation is as:

```
....
```

```

....
....
assign_weight(root of expression tree) ;

expand_expr();      /* Modified expand_expr that will traverse as per the weights*/
....
....
....

```

3. Traversing the tree in HSTF fashion.

Now, we have an annotated tree.

This scheme works only for arithmetic expressions involving add,sub,mul,div, modulo, etc

So, we do the following:

1. In expand_expr, we look for nodes of type add (denoted as PLUS_EXPR), sub, mul, etc.
2. For such cases, we look at the weights & then the heavier branch is taken.

This works as:

```

expand_expr()
{
...

if(type of node is plus, minus, add, etc)      /* Any binary operation */
{
    if(left node is heavier) {                /* First left then right */

        expand_expr(left node);
        expand_expr(right node);

    }
    else{                                       /*right is heavier. first right, then left*/

        expand_expr(right node);
        expand_expr(left node);

    }
}
}
}

```

Thus, GCC is optimized.

This new option is '-freg-eff'

```
gcc -freg-eff hello.c      /* compiling with this optimization */
```

Proving the correctness

Now, the optimization is done. But, for the optimization to be accepted, we must show that the optimization did not introduce any new bugs.

We need to show that, no matter what the input program, our optimization gives the same result as the unoptimized code.

The improvement is better than O2 as well. That is, (O2 + HSTF) is better than (O2 alone). See [BABU] for complete implementation details of the IITB patch.

Our Job

Thus, it has been demonstrated that HSTF is indeed a better algorithm than LSTF. However, Abacus is a non-standard architecture. Apart from this, since it is a processor manufactured by the DRDO (Defence Research and Development Organization), it is a classified architecture. Our job was to investigate the application of the HSTF algorithm for GCC on Intel Pentium IV architecture.

3.5 Register Usage Prediction

Optimizing Pentium 4

What happens if we apply the same optimization to Pentium 4 ?

One of the changes is that labeling of nodes is non-trivial. Here, we have memory operands as well. The logic used is that, suppose I have a right heavy tree. Even if I label the nodes as per Abacus strategy, I should get a somewhat better performance than pure LSTF.

This is what they did with the IITB patch. They applied the same patch to Intel Pentium 4, then ran the test suite of programs such as gzip, bzip2, parser, etc on the P 4. They noticed a performance gain.

Prediction Strategy

We also propose to develop an algorithm for predicting the register usage and the stores performed by a given expression. Let us now, look into the reasons for doing so.

The IITB patch for Abacus, uses only register usage as a metric for assigning weights. This is not proper, as we shall see in the later sections.

The HSTF algorithm discussed earlier, is theoretically correct. Hence, if it does not work for Pentium IV as expected, then something must be wrong with the implementation, and not the algorithm.

We think that the fault lies in the metric considered for the purpose. Considering only the register usage of the expression tree cannot give us the complete measure of the heaviness of the tree. This may be the cause of HSTF failure on Pentium IV. (Causes of failure are discussed in detail in the next chapter)

We propose to use a combination of

- Register usage
- The stores (register to memory moves) performed
- The depth of the tree node

as an improved metric to determine the heaviness of tree nodes. We propose to devise and implement algorithms for this purpose.

3.6 The GCC Code Analysis Tool

We also propose the development of the GCC code analysis tool. The proposed tool must include features such as:

Expression Testing: A way to build C code files from expressions, compile those files, compare actual as well as cycle timings for files.

Assembly Optimizations: Implementations of all the assembly optimizations that we found, so that the effect of the optimizations may be easily seen and analyzed.

Information: The tool shall include detailed FAQs and HOWTOs describing various aspects of the project. This will serve as a guide to future research in the area.

Other Utilities: The tool shall also include other utilities such as register usage counters, code transformers, etc. that make the understanding and analysis of code easier.

We chose PHP as the language to implement the tool. The reasons for this are as follows:

- PHP is an open source technology that runs on all kinds of systems
- PHP allows users to invoke system calls. This is useful in compiling programs, timing them, etc.
- Familiarity of working in PHP. So, the implementation could begin without much of a learning effort.
- Easy to create user interfaces in HTML

3.7 Data Flow Diagrams

The Data Flow Diagrams (DFDs) are a way to describe a system in a graphical manner. In this section, we describe our project as a series of DFDs.

Level 0 DFD

This DFD presents a single bubble view of the system functionality. It is illustrated in the figure below.

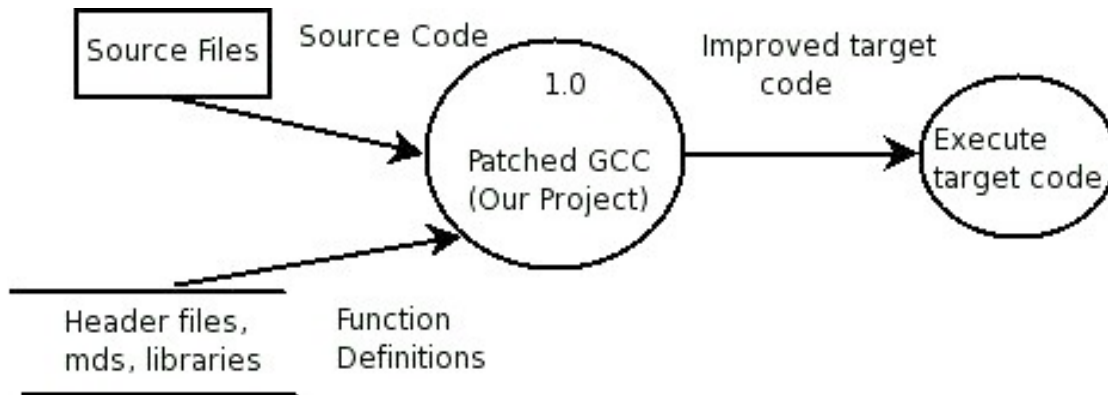


Figure 3.3: Level 0 DFD of the project.

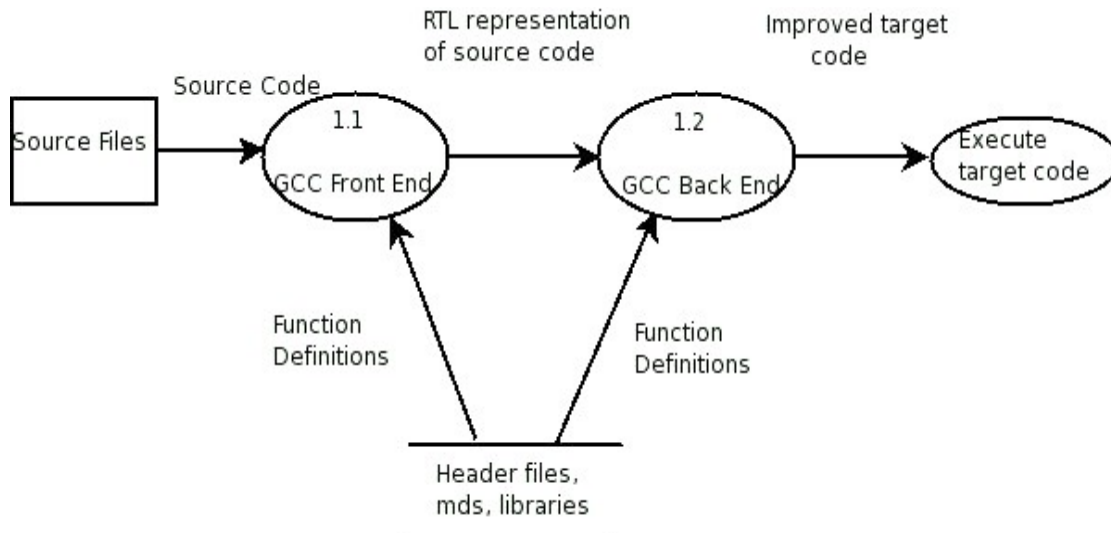


Figure 3.4: Level 1 DFD of the project.

Level 1 & 2 DFDs

The Level 1 & Level 2 DFDs split the single bubble in Level 0 into further, more finer representations of system functionality.

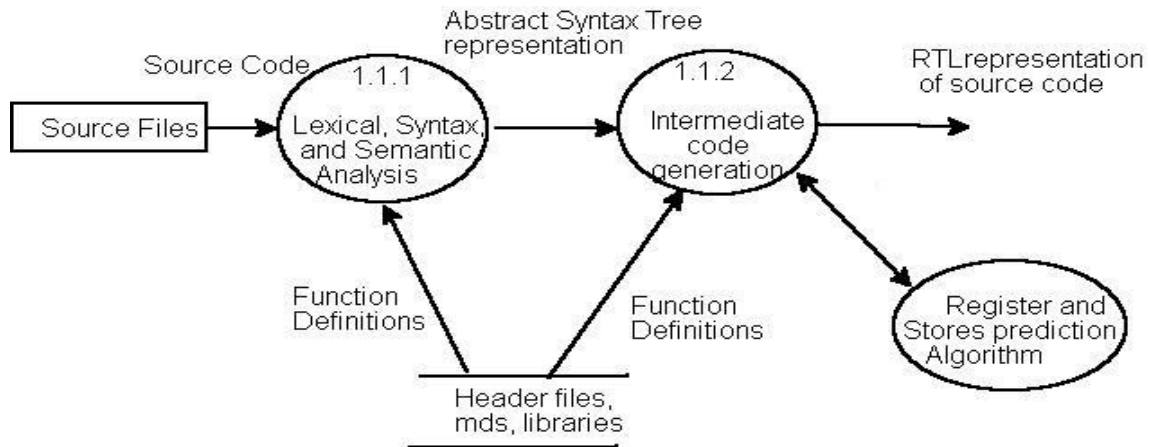


Figure 3.5: Level 2 DFD (Part 1) of the project.

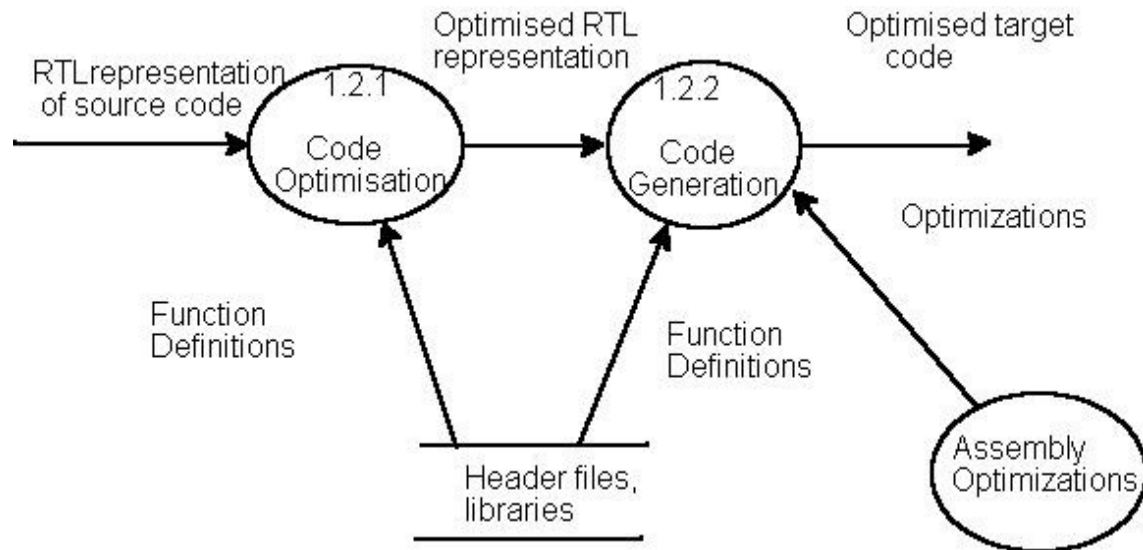


Figure 3.6: Level 2 DFD (Part 2) of the project.

4. System Implementation

In this chapter, we discuss the actual implementation of our project. We discuss all the underlying algorithms, implementation details and features of our system.

4.1 Problems in HSTF implementation

While implementing HSTF for GCC on Intel Pentium IV machines, we faced some problems. These problems prevented us from actual implementation of HSTF. We discuss these problems in this section.

1. Introduction

A patch to GCC called as the "IITB patch" has been developed at IITB. This patch implements the HSTF algorithm for GCC. In short, it forces GCC to use HSTF instead of LSTF. But, this change is not universal. LSTF traversal is changed to HSTF traversal only for some cases. These include arithmetic expressions, arrays, etc.

Now, the IITB patch was designed for a processor called ABACUS. ABACUS is a RISC processor developed by the Defence Research and Development Organization(DRDO). However, as the area of operation of the patch is in the GCC front end, the patch has been applied to Intel Pentium IV architecture. This shows improved results as compared to the un-patched GCC. One task of our project was to improve the patch performance on Pentium IV architecture.

2. Why improve the existing patch?

One thought that may occur to you is that why does the patch not work as it is for Pentium IV ?

The answer is that though the patch is a part of the GCC front end, it relies on certain architectural features such as instruction set, registers, etc. The architecture of Pentium IV being very different from that of ABACUS, the patch does not give optimum performance. Some aspects of improper performance of the patch are as follows:

- i. The patch considers "CPU registers used" as the only metric for determining the heaviness of an expression tree. This is inaccurate in case of Pentium IV.
- ii. The patch considers only immediate and register as the possible type of operands. On Pentium IV machines, memory operands also exist and need to be considered.
- iii. The patch treats all instructions equally. However, in case of Pentium IV, a division is much more costly than an addition.

The patch makes the above assumptions as it is designed for a RISC machine. Thus, the patch needs to be improved for Pentium IV.

We found that HSTF implementation on Pentium IV, was not as easy as expected, because of the following reasons:

- If the order of execution is changed from the usual LSTF, then GCC performance deteriorates inexplicably.
- GCC starts performing unnecessary stores & the patched GCC (with HSTF) performance is either worse than or equal to the original GCC performance.
- We think that this problem may be addressed by a better metric of heaviness. We have proposed a better metric consisting of register usage, tree depth and loads and stores performed.

4.2 The Prediction Algorithms

We constructed prediction algorithms, that predict register usage, and the number of stores performed by a given expression. In this section, we discuss these algorithms.

The possibilities

There are several cases that need to be considered in order to arrive at a concrete decision about register usage & stores prediction. The possibilities can be illustrated as follows:

Given a node, there are various possible types of operands, types of parent node, and position in tree, that need to be considered while making a prediction.

Syntax: <Left Child><Right Child>

Cases:

1. IMM, IMM (Immediate)
2. VAR, VAR (Variable)
3. OPER, OPER (Operator)

4. IMM, VAR
5. VAR, IMM

6. IMM, OPER
7. OPER, IMM

8. VAR, OPER
9. OPER, VAR

All these cases have to be repeated for:

1. The current node is root node
2. The current node is left child of its immediate parent
3. The current node is right child of its immediate parent

Also, the possible values of each constituent that need to be considered:

IMM: 1, 2, 3, powers of 2, non-powers of 2, 23, 50, and other random numbers
OPER: ADD, SUB, MUL, DIV
VAR: This can be any variable. (Just one case)

We considered several cases of each type before formulating the algorithm.

Syntax

The sample syntax used in these algorithms was as follows:

current = current node.
parent = immediate parent of current node.
left(current)= left child of current node.
right(current)= right child of current node.
reg_usage(current)=register usage during evaluation of tree rooted at current node.
stores(current) = stores needed during evaluation of tree rooted at current node.

ADD_OPER = Addition Operator
SUB_OPER = Subtraction Operator
MUL_OPER = Multiplication Operator
DIV_OPER = Division Operator

A sample algorithm

Now, we present a sample of the algorithms we developed. This is a piece of algorithm for the multiplication operator.

1. if parent is null

/* Both children are VAR */

1.1 if left(current) is VAR and right(current) is VAR
then reg_usage equals 1

/* Left child is IMM right is VAR */

1.2 if left(current) is IMM and right(current) is VAR and left(current) is power of 2
then reg_usage equals 1

In 1.1, both the children of the current node are variables. In this case the register usage is one. Such pattern is repeated for all the nodes.

4.3 The Agnel GCC Group Patch

The Agnel GCC Group patch is the implementation of the above discussed algorithm into the GCC source. For this purpose, we introduced some new code into GCC. Some of the code is explained, as well as other related issues are discussed in this section.

The GCC options

The Agnel GCC Group patch is not enabled default. There are two options provided, that are to be explicitly invoked. These are declared as:

Flags

The flags we added are off by default. They have to be activated explicitly by giving command line options. The declaration of these flags is as follows:

```
extern int flag_op_behave;                /* The main option */  
extern int flag_acc_tdepth;             /* The option for taking tree depth into account*/
```

This code is added to the GCC source file flags.h Note that the tree depth option does not work on its own. It has to be specified along with the first option.

Why two flags ?

The first flag is used for providing basic prediction functionality. The second flag, when enabled performs another pass over the tree. If a user wants to just assign weights to every node, then accounting for tree depth is not required. Also, the problem of tree depth causing differences in usage is caused by the LSTF nature of traversal. Hence, the need to implement two options, instead of one.

Flag Descriptions

This code is added into the file toplev.c The flags and their description are as follows:

```
int flag_op_behave=0;  
-  
  
-int flag_acc_tdepth=0;  
-  
- { "op-behave",&flag_op_behave,1,  
- N_("Predict operator behaviour on per-node basis for MADS (*,+/, -)")}.  
- }
```

New Data Structures

The existing data structures used in GCC, do not have provision to store values such as register usage, node number, stores performed, etc. We added these data items to the existing GCC node structure in the file tree.h These data items are as follows:

```
int stores;  
int reg_usage;
```

```
int node_number;  
int right_heaviness;
```

Macros

In order to retrieve and set the values of the new variables, that we have defined, we also define some macros. These are also added to the file tree.h. Some sample macros are presented below:

```
#define SET_REG_USAGE(ptr,ru) ((ptr)->common.reg_usage=ru)  
##define GET_REG_USAGE(ptr) ((ptr)->common.reg_usage)  
-  
##define SET_STORES(ptr,st) ((ptr)->common.stores=st)  
##define GET_STORES(ptr) ((ptr)->common.stores)
```

The Function Descriptions

The algorithms that we implemented, have been included into the GCC source code in the file stmt.c. For each algorithm, we defined a separate function. The function declarations are as follows:

```
-void predict_oper_behave(tree current,tree* parent,int node_num);
```

This is the initial function called, when a program is compiled with either one or both of our options.

```
-void predict_with_tree_depth(tree current);
```

This function is used for performing predictions, by taking tree depth into account.

```
--void mult_behave(tree current,tree* parent,int node_num);
```

```
-void sub_behave(tree current,tree* parent,int node_num);
```

```
-void add_behave(tree current,tree* parent,int node_num);
```

```
-void div_behave(tree current,tree* parent,int node_num);
```

These are the four prediction algorithms for the specified operators.

```
- void print_oper_behave(tree current);
```

This function prints out the predictions for every node. It is called after the prediction algorithms have done their job.

```
--void oper_type(tree current);
```

This function is invoked by the print_oper_behave for displaying the type of a given node. Type may be either variable, operators, etc.

```
-int level(int node_num);
```

This function accepts as parameter the node number, and from that it calculates the level in the tree at which the node is present.

```
-int code_verify (current)
```

We do not consider all kinds of assembly code. We consider only certain opcodes such as plus, minus, etc. This function filters out the unwanted opcodes.

About the Node Numbering

The nodes are numbered as follows:

$2*n - 1$ for left child of the current node & $2*n + 1$ for right child of the current node

Assuming that, 'n' is the current node number.

Invoking the patch

To invoke the patch, whenever the option is specified, we added code to the function “init_stmt_for_function“, in the file stmt.c The way, the function is called, is illustrated by the code below.

```
if(flag_op_behave && code_verify(exp)){  
  
    tree* parent=NULL;  
    int node_number=0;  
  
    predict_oper_behave(exp,parent,node_number);  
  
    if(flag_acc_tdepth) {  
  
        predict_with_tree_depth(exp);  
    }  
    print_oper_behave(exp);  
}
```

Working of the patch

Here, if the option is enabled, the predict_oper_behave function is called. If the tree depth option is enabled as well, then the predict_with_tree option is called. These two function call the prediction algorithms, as needed. At the end of the prediction, the print_oper_behave function is called, that prints the results. This cycle is repeated for every node.

GCC Functions and Macros used

TREE_CODE: This gives the opcode of the current node.

TREE_OPERAND: Pointer to the operands of an operator node.

Accounting for tree depth

The second option, for accounting for tree depth, has some special cases that need consideration. Following are some cases where the tree depth option succeeds and fails.

Fails:

$((a+b)/((c+d)/((e+f)/((g+h)/((i+j)/((k+l)/((m+n)/((o+p)/((q+r)/((s/t)/((u-v)/((w+x))))))))))))))$

Additional store for (s/t) due to the '+' operator in (w+x). This phenomenon is observed only if the operator is addition. In case of the rest of the operands, subtraction for example, no stores are observed.

Match:

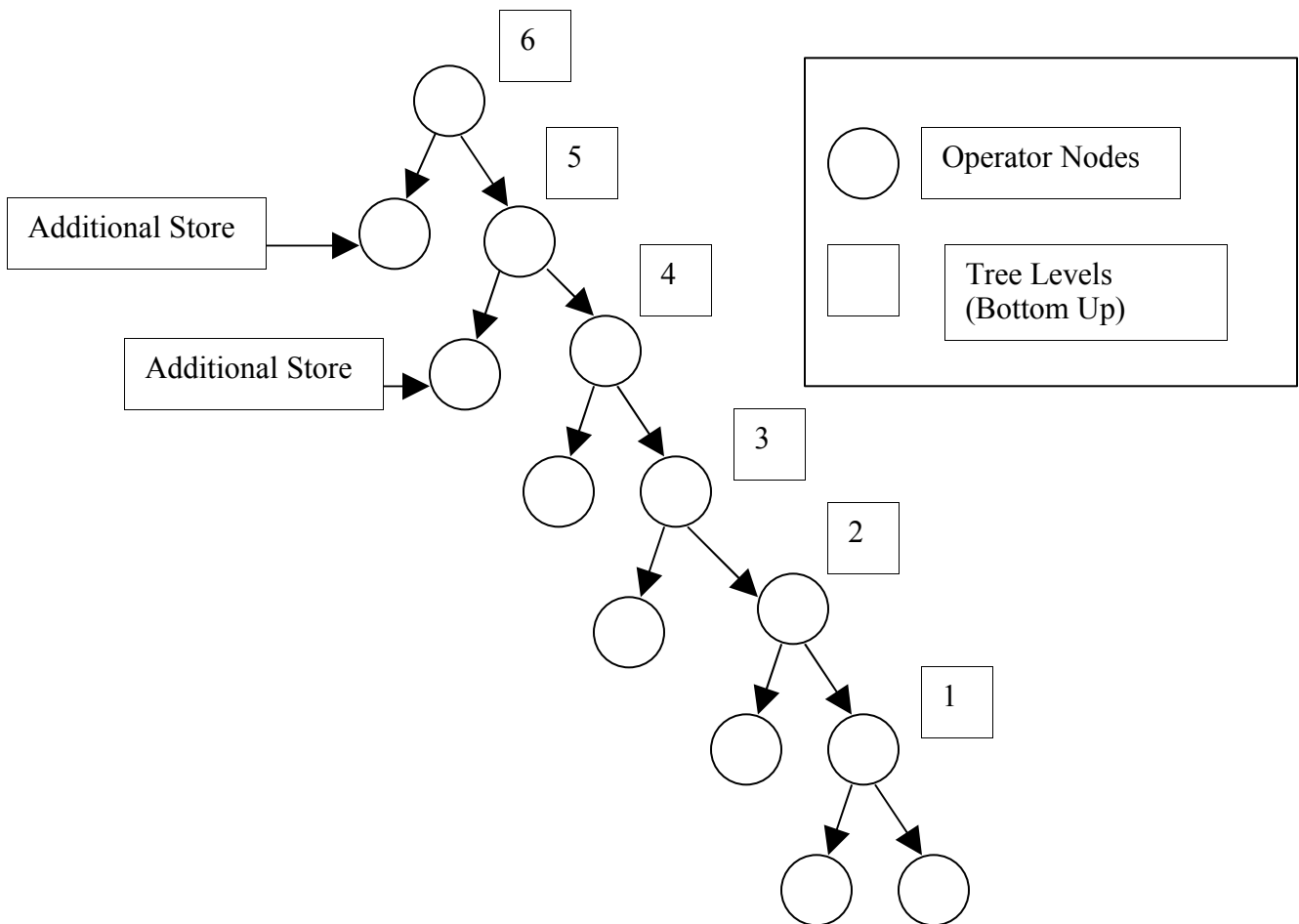
$((a+b)/((c+d)/((e+f)/((g+h)/((i+j)/((k+l)/((m+n)/((o+p)/((q+r)/((s/t)/((u-v)/((w-x))))))))))))))$

Match:

$((a+b)/((c+d)/((e+f)/((g+h)/((i+j)/((k+l)/((m+n)/((o+p)/((q+r)/((s*t)/((u-v)/((w-x))))))))))))))$

A Unique Phenomenon

A strange phenomenon is observed, as illustrated in the figure below. For a tree more than four levels deep, this phenomenon is observed. If we start counting from the bottom-up, then the left children of the nodes numbered five or higher, perform stores.



4.4 Assembly Optimizations

We have found four avenues for improving the assembly code generated by GCC. They are listed as follows:

1. Redundant moves & stores
2. Replace LEAL (used for addition) by ADDL , ADDL/MOVL
3. Replace LEAL (used for loading the effective address) by MOVL
4. Replace SUBL by NEGL and ADDL

1. Redundant moves & stores

GCC performs several unnecessary moves. These moves can be **register to register** or **register to memory**. We propose these moves and stores to be replaced by equivalent efficient code.

2. Replace LEAL (used for addition) by ADDL , ADDL/MOVL

This is a case of better instruction selection. GCC uses the LEAL instruction to perform additions of the form: $a = a + b$ and $a = b + c$. LEAL does both these jobs in a single instruction. However, this is inefficient as compared to using the simpler ADDL & MOVL instructions.

Sample Case:

TYPE 1

Original sequence : `leal (%ebx,%ecx), %eax`

Optimized sequence :
`addl %ebx, %ecx`
`movl %ecx, %eax`

Here, the problem is that content of 'ecx' is changed, which may not be desirable in all cases. If 'ecx' is used in future, then this optimization may fail.

TYPE 2

Original sequence : `leal (%ebx,%ecx), %ecx`

Optimized sequence :
`addl %ebx, %ecx`

TYPE 3

Original sequence : `leal (%ebx,%ecx), %eax`

Optimized sequence : `movl %ecx, %edx`
`addl %ebx, %edx`
`movl %edx, %eax`

To implement this optimization, we need to find a free register, to perform a move operation. `edx` is the register used in the above case.

3. Replace LEAL (used for loading the effective address) by MOVL

This is a case of better instruction selection. GCC uses the LEAL instruction to load the effective address of a given operand. The operation is then performed on the actual location of the operand. A better way is to load the value of the operand in a register, perform the operation on the value & store back the value to the operand location.

This is based on the premise that register operations are cheaper than memory operations.

Sample Case:

TYPE I

Original sequence : `leal -208(%ebp), %ebx`
`sarl $31, %edx`
`idivl (%ebx)`

Optimized sequence : `mov -208(%ebp), %ebx`
`sarl $31, %edx`
`idivl %ebx`

4. Replace SUBL by NEGL and ADDL

This is a case of better instruction selection. GCC uses the SUBL instruction to perform subtractions. It has been observed that negation followed by addition is more efficient than the SUBL instruction.

Sample Case:

Original instruction : `subl -208(%ebp), %eax`

optimized sequence : `movl -208(%ebp), %edx`
`negl %edx`
`addl %edx, %eax`

About SUBL optimization

One point to note about the SUBL optimization is that, SUBL is technically worse than the combination of NEGL and ADDL. However, this is not observed practically. We believe that the Pentium IV pipeline may be affecting this. If the pipeline optimizations are performed, then this optimization will surely work.

4.5 GCC Tool features

In this section, we discuss the important features of the GCC code analysis tool.

Register Usage Counter

The Register Usage Counter checks whether any of the six user accessible registers (eax, ebx, ecx, edx, esi, edi) are used in a given assembly code file. It also counts the number of times each register has been used.

Assembly Optimizations

The GCC tool automates several optimizations. The various assembly optimizations performed are as follows:

- Redundancy Removal
- Replacing SUBL by NEGL & ADDL
- Replacing LEAL (Addition) by ADDL & ADDL with MOVL
- Replacing LEAL (Effective Address) by MOVL

Expression Tester

The expression tester automates the following tasks:

- Creation of C source files from expressions.
- Compilation and timing comparison of various files.
- Facility for specifying GCC prefixes, loop counters, etc.

5. Performance Measurement

This chapter illustrates the improvements achieved by our project. It provides numerical evidence of the prediction accuracy of the GCW patch, i.e. what percentage of time does the prediction of the patch match the actual register usage and stores of the expression. In case of assembly optimizations, timing comparisons are provided to prove that the proposed optimizations are indeed better than the currently produced code. Screen shots of the GCC code analysis tool, show the tool in action.

5.1 Prediction Accuracy

The GCW patch was constructed by evaluating the results of around 300 experiments involving all kinds of operator combinations. We conducted 25 test cases on the patch to verify its accuracy after it was integrated into GCC. This section illustrates those test cases.

5.2 Optimization Timings

We have three types of optimizations. In this section, we illustrate the improvements with some statistics.

Type I: Replace LEAL (used for addition) by ADDL , ADDL/MOVL

Type II: Replace LEAL (used for loading the effective address) by MOVL

Type III: Replace SUBL by NEGL and ADDL

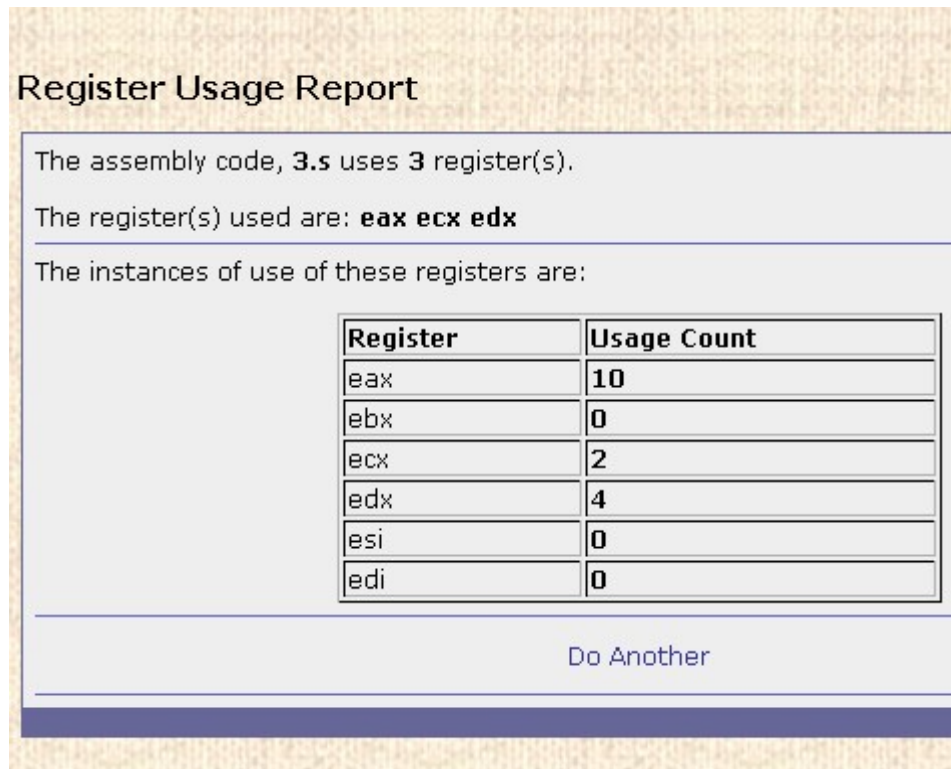
Serial No.	Type of Optimization	Case	Total Cases	Improvement Observed in (Cases)
1	1	1	12	9
2	1	2	12	8
3	1	3	12	12
4	2	1	15	12
5	2	2	15	9
6	2	3	12	7
7	2	4	12	9
8	3	1	12	9
9	3	2	12	9
10	3	3	12	7
11	3	4	12	8

5.3 GCC Tool screen shots

To illustrate the working of the GCC code analysis tool, we present some screen shots in this section. These screen shots highlight some of the functionalities of the tool.

1. Register Usage Report

The register usage report is a statement of the total number of registers used in an assembly file. The report also counts the instances of specific registers being used.



Register Usage Report

The assembly code, **3.s** uses **3** register(s).

The register(s) used are: **eax ecx edx**

The instances of use of these registers are:

Register	Usage Count
eax	10
ebx	0
ecx	2
edx	4
esi	0
edi	0

[Do Another](#)

Fig 5.1: Register Usage Report

2. Main Control Panel

This is the main control panel of the tool. The tool uses a framed layout and hence, the control panel remains on the left. It provides links to all functionalities of the tool.

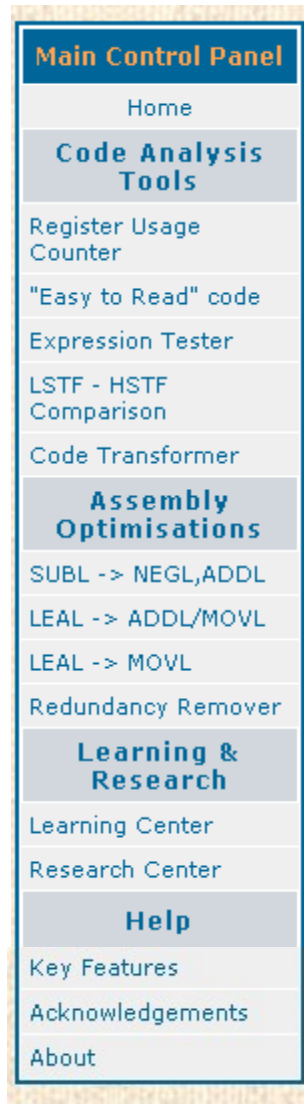


Fig 5.2: Main Control Panel

3. Optimization Input

This is the usual input screen for an optimization. All the user needs to do is to enter the name of the assembly code file

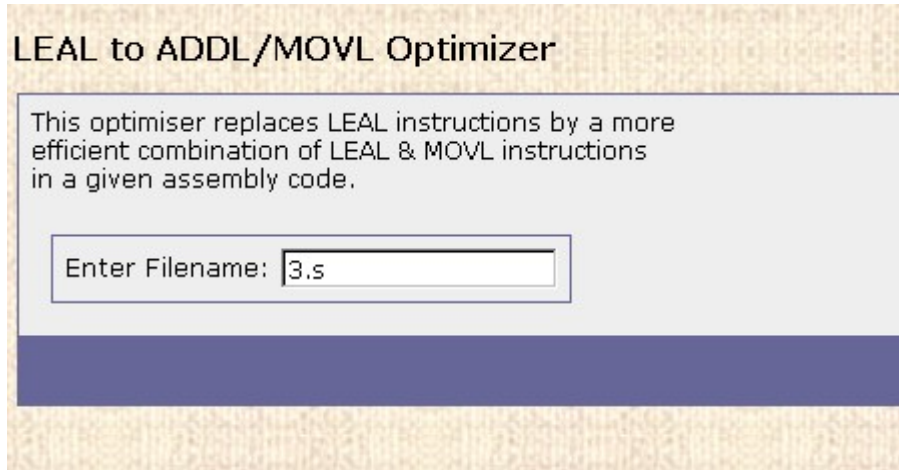


Fig 5.3: Optimization Input Screen

4. Optimization Result

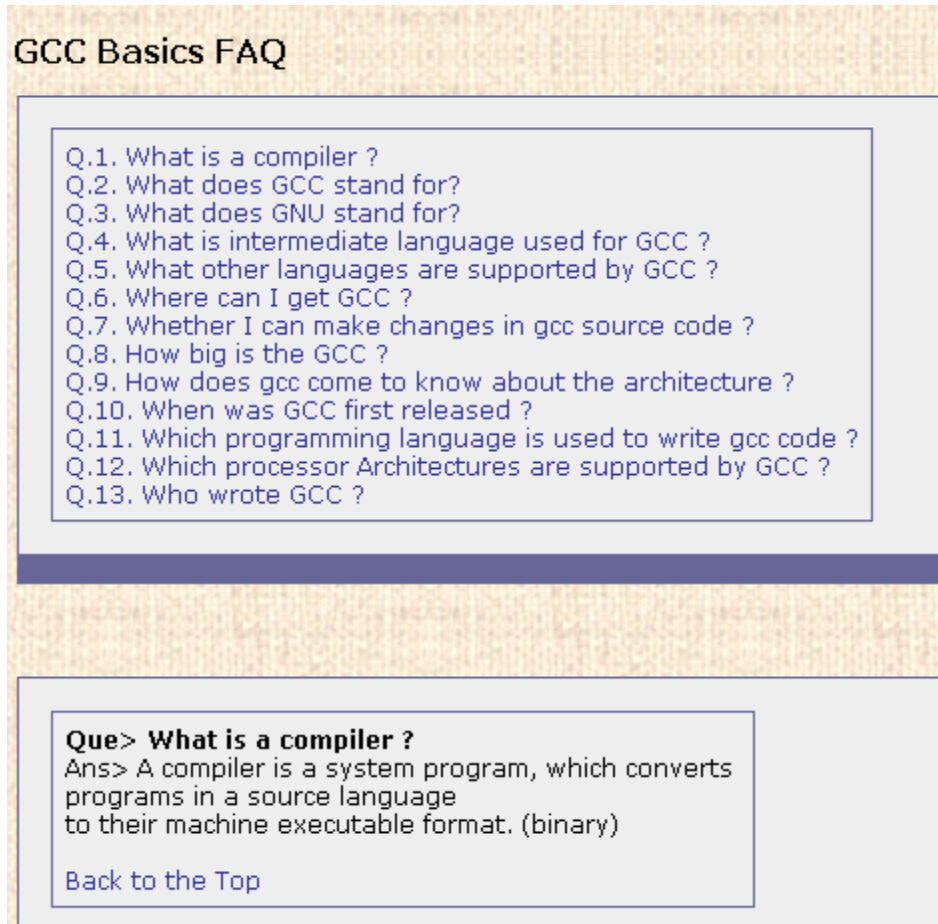
The following screen shot shows the result of the `SUBL -> NEGL + ADDL` optimization. The unoptimized portion is marked red and the optimized one is marked blue.

<code>shrl \$4, %eax</code>	<code>shrl \$4, %eax</code>
<code>sall \$4, %eax</code>	<code>sall \$4, %eax</code>
<code>subl %eax, %esp</code>	<code>negl %eax</code> <code>addl %eax, %esp</code>
<code>movl \$1, -24(%ebp)</code>	<code>movl \$1, -24(%ebp)</code>
<code>movl \$1, -20(%ebp)</code>	<code>movl \$1, -20(%ebp)</code>

Fig 5.4: Optimization Result

5. FAQs

This is a screen shot of the GCC basics FAQ. Many more such FAQs form a part of the tool.



GCC Basics FAQ

- Q.1. What is a compiler ?
- Q.2. What does GCC stand for?
- Q.3. What does GNU stand for?
- Q.4. What is intermediate language used for GCC ?
- Q.5. What other languages are supported by GCC ?
- Q.6. Where can I get GCC ?
- Q.7. Whether I can make changes in gcc source code ?
- Q.8. How big is the GCC ?
- Q.9. How does gcc come to know about the architecture ?
- Q.10. When was GCC first released ?
- Q.11. Which programming language is used to write gcc code ?
- Q.12. Which processor Architectures are supported by GCC ?
- Q.13. Who wrote GCC ?

Que> What is a compiler ?
Ans> A compiler is a system program, which converts programs in a source language to their machine executable format. (binary)

[Back to the Top](#)

Fig 5.5: GCC basics FAQ

6. Conclusions & Future Work

6.1 Conclusions

At the end of the project, our work can be summarized as follows:

- We studied the Implementation of HSTF for GCC on Intel Pentium IV
- We wrote an algorithm to predict the register usage and the number of stores taken by an arithmetic expression
- We found four assembly optimizations to improve the quality of existing code.
- We constructed a tool in PHP to analyze code produced by GCC, implement optimizations and auto generate C files from input expressions.

6.2 A Better Metric

We have concluded that register usage is not the only indicator of heaviness of a tree. Because of this, GCC performs poorly on the Intel Pentium IV processors. We think that considering a better metric, reflecting all the factors that affect tree traversal, will remedy this. The better metric should account for things like:

- The number of loads and stores performed
- The relative position of a node with respect to its immediate parent node
- The depth of the node
- The cost of each operation as perceived by GCC

Some, as yet undiscovered factors may also be affecting the process. Hence, the quest for a better metric remains a future task.

6.3 Improved Instruction Selection

We have been able to find three instances of instructions doing a better job than the ones currently used by GCC. Much more research is needed to find all possibilities of improvement in this area.

6.4 Extending the GCW Patch

The GCW patch, at the moment deals only with the four basic arithmetic operators viz., addition, subtraction, multiplication, and division. It needs to be extended in the following ways:

- Include other operators such as bitwise operators, and logical operators.
- Improve the current performance of the patch, so that it predicts correct results all the time.
- Some research is needed into the behavior of immediate operands, to ascertain their exact usage patterns.

6.5 Machine Description for Pentium IV

At present, there is no MD (Machine Description) specifically for Pentium IV. This prevents utilization of Pentium IV specific features by GCC, and may result in a lesser performance. This shortcoming needs to be addressed by creating an MD for Pentium IV.

Appendix A.1 GCC Installation

About this Appendix

This appendix describes the steps required to configure, build and install gcc. The options provided in this chapter are those which were required during our project work. The appendix in no means is a complete reference to the above mentioned task and should be used in conjunction with the official install document found in the INSTALL subdirectory of the gcc source directory.

This typeface is used for the commands that have to be typed in the shell.

GCC version used for installation – 3.4.3

Installation notes

Assuming present working directory is \$GCC_HOME

1. Obtain the gcc source

2. Unpack the source to get gcc-3.4.3

```
tar -xvzf gcc.tar.gz - for gz files
OR
tar -xvjf gcc.tar.bz2 - for bz2 files
The output will be a gcc-3.4.3 directory
```

3. Make two more directories gcc-3.4.3-obj and gcc-3.4.3-ins; the object and installation directories

```
cd gcc-3.4.3-obj
cd gcc-3.4.3-ins
```

4. Configuration

```
cd gcc-3.4.3-obj
../gcc-3.4.3/configure --program-
prefix=mygcc --prefix=$GCC_HOME/gcc-3.4.3-
ins --enable-languages=c,c++
```

Note:- There are two hyphens (-) before program-prefix. Enable-languages and prefix and the above is a single line command

--program-prefix now will add the prefix, in this case, **mygcc-** to the program names in the gcc installation bin directory. This allows your previous existing gcc installation to work without any changes

Assuming **gcc** refers to (if any) previous installation of gcc, by adding the prefix **mgcc-**, the newly installed gcc can be used in conjunction with the old as given in the example

```
gcc temp.c
my-gcc temp.c
```

The **--prefix** option provides path for installation directory

The **--enable-languages** option species the languages that the compiler will accept

5. Building

The make command should do the trick

```
make
```

6. Installation

Now for the final installation

```
make install
```

7. Setting the path

To include the newly created bin files in the gcc-3.4.3-ins bin/ directory in the PATH

```
cd
vim .bashrc
add PATH="$PATH:$GCC_HOME/gcc-3.4.3-ins"
exec bash
```

8. Testing

```
mygcc-gcc --version
```

This should print out the appropriate gcc version.

Appendix A.2 Patching Basics

This appendix describes the process of applying the GCW patch to the GCC version 3.3.3 you have installed.

Applying the patch to GCC

For patching gcc with patch file named **mypatch**

1. Make a directory which will have the gcc-3.3.3.tar.gz and the patch

example

```
#mkdir test
```

copy the gcc tar and patch file into test directory

2. unzip the file

```
#tar -xvzf gcc-3.3.3.tar.gz
```

3. Verify this directory structure

```
test/gcc-3.3.3/
```

4. Patch the gcc

if already not inside test

```
#cd test
```

```
#patch -p1 <mypatch
```

verify the four files namely,

```
stmt.c
```

```
flags.h
```

```
toplev.c
```

```
tree.h
```

have been patched successfully

5. Other directories

```
#mkdir obj
```

where obj is on the same level as gcc-3.3.3 ie inside test

6. Configure

```
#cd obj  
#./gcc-3.3.3/configure --enable-languages=c --prefix=./ins
```

7. make process

```
#make ; make install
```

8. If above complete without errors obj/ins have the binaries

```
# cd ins/bin
```

9 Testing

```
#./gcc --op-behave a.c
```

References

[JAIN] Issues in re-targeting GCC, An M.Tech. Dissertation by Nitin Jain, CSE IITB, 2004.

[BABU] Improving machine description of Abacus, M.Tech. Project Report (II) by Rajesh Babu, CSE IITB (<http://cse.iitb.ac.in/~babu>),

[GOUG] An Introduction to GCC, By Brian J. Gough, foreword by Richard M. Stallman, Network Theory Limited, 15 Royal Park, Bristol BS8 3AL, United Kingdom (<http://network-theory.co.uk/gcc/intro/>)

[BODI] Lectures in CS536, Rastislav Bodik, CS Division, UC Berkeley (<http://www.cs.berkeley.edu/~bodik.html>)

[AGNE] How to Optimize for the Pentium family of microprocessors, by Agner Fog, Ph. D.

[WOOD] Woodmann website (<http://www.woodmann.com/fravia/hackmo1.htm>)

Bibliography

This bibliography includes a list of resources related to this report.

GCC Source

- The GCC website (<http://gcc.gnu.org/>)

GCC Manual / Books

- Using & Porting the GNU Compiler Collection (<http://gcc.gnu.org/>)
- An Introduction to GCC, By Brian J. Gough, foreword by Richard M. Stallman, Network Theory Limited, 15 Royal Park, Bristol BS8 3AL, United Kingdom (<http://network-theory.co.uk/gcc/intro/>)

Compiler Theory

- Compilers: Principles, Techniques, and Tools Aho, Sethi, Ullman, Pearson Education
- Lectures in CS536, Rastislav Bodik, CS Division, UC Berkeley (<http://www.cs.berkeley.edu/~bodik.html>)
- Lecture Notes of Chau-Wen Tseng (CMSC 430), Univ. of Maryland, College Park (<http://amber.cs.umd.edu/class/430-f03/>)

GCC Project at IIT Bombay

- Issues in re-targeting GCC, M.Tech. Dissertation by Nitin Jain, CSE IITB
- Improving machine description of Abacus, M.Tech. Project Report (II) by Rajesh Babu, CSE IITB (<http://cse.iitb.ac.in/~babu>)
- IIT Bombay website (<http://iitb.ac.in>)

GNU Project / Philosophy

- The GNU website (<http://gnu.org/>)
- GNU Philosophy(<http://www.gnu.org/philosophy/philosophy.html>)
- Free Software Foundation (<http://www.fsf.org/>)

- Free as in Freedom: Richard Stallman's Crusade for Free Software By Sam Williams O'Reilly & Associates, Inc. (<http://www.faifzilla.org>)
- Richard Stallman's website (<http://www.stallman.org>)

Intel C++ Compiler (ICC)

- Intel (<http://www.intel.com/>)
- The Intel C++ Compiler for Linux(<http://www.intel.com/cd/software/products/asmo-na/eng/compilers/219623.htm>)

Intel Pentium IV processor

- Intel Developer's site (<http://developer.intel.com/>)
- Tom's Hardware (<http://www.tomshardware.com/>)
- Advanced Microprocessors, by Daniel Tabak, Tata McGraw Hill
- The Intel Microprocessors: Architecture, Programming and Interfacing, by Barry Brey.
- Pentium Processor System Architecture, by Tom Shanley, Addison Wesley Press

Role of EAX in expression evaluation

- Woodmann website (<http://www.woodmann.com/fravia/hackmo1.htm>)