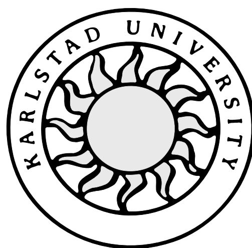


Computer Science

Peter Nenzén, Anders Rågård

Tail Call Elimination in GCC

Bachelor's Project

2000:12

Tail Call Elimination In GCC

Peter Nenzén, Anders Rågård

© 2000. The authors and Karlstad University

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Peter Nenzén

Anders Rågård

Approved, 2000-06-09

Advisor: Donald F. Ross

Examiner: Stefan Lindskog

Abstract

This project is initiated by Ericsson Infotech AB, Department of Test, Support and Simulated Platforms (TSP) in Karlstad. At TSP, simulation of telephone switches is performed. When simulating, gotos are used in the simulation language (C). Newer versions of the C compiler, GCC, do not support the use of gotos between functions. Ericsson needs to develop support for a technique called “tail calls” in order to replace the code using gotos. This project involves investigating the requirements for GCC to support the mechanism for handling tail calls. As a background, the authors will describe in general terms, the function and phases of a compiler, and in particular those of GCC.

We have found some related projects that suffer from the lack of tail call elimination. A person involved in one of these projects has worked together with us trying to solve the common problem.

A single user can make a difference to the development of GCC. This project has awoken the interest for a tail call solution. We assert that such a solution has emerged much earlier due to our actions, than it would have done else.

When a solution was published, our project changed direction from designing a solution to evaluating an already implemented solution.

Ericsson’s goal was not achieved one hundred percent. However, this project has brought Ericsson much closer to a solution that would handle a related problem, that of indirect tail calls. While finalizing this report, we conferred with our knowledgeable contacts in order to investigate how much work that remains in order to adjust the existing GCC version, (with the Cygnus/Jelinek patches committed) to solve Ericsson’s tail call problem.

The authors recommend Ericsson to use the outcome of this report as a springboard for further investigations and development, if Ericsson intends to develop an in-house solution.

or

to purchase a solution from one of the various companies that supplies GCC support and hence are possible contractors for the remaining work.

Contents

1 Introduction ... 1

1.1 Defining the Problem ... 1

1.1.1 Background: ... 1

1.1.2 Assignment: ... 2

1.2 Scope of the Assignment ... 2

1.3 Reading This Report ... 2

2 Background ... 5

2.1 Introduction ... 5

2.2 Ericsson, As the Company Describe Itself ... 5

2.3 Project Overview ... 6

2.4 Tail Calls ... 7

2.5 The Functionality of a Compiler in General ... 7

2.5.1 The Symbol Table Manager ... 9

2.5.2 Error Handler ... 9

2.5.3 The Analysis Phases ... 9

2.5.4 Generation of Intermediate Code ... 10

2.5.5 Optimization ... 10

2.5.6 Generation of Target Code ... 10

2.5.7 Example of Translation of a Statement ... 11

2.6 The Compiler System GCC ... 12

2.6.1 The Passes of GCC ... 13

2.7 Optimization Options in GCC ... 16

2.8 Platform ... 16

2.9 Chapter Summary ... 17

3 The Task ... 19

3.1 Introduction ... 19

3.2 Beginning the Task ... 19

3.3 The Problem ... 20

3.4 Related Projects ... 24

3.4.1 The School of Computer Science in Toronto ... 24

3.4.2 Jacub Jelinek ... 25

3.4.3 Marc Lehman and Per Bothner ... 25

3.4.4 Jeffrey A. Law, the GCC Release Manager ... 26

3.4.5 PhD Markus Pizka at Microsoft Research in Cambridge, England 27
3.5 The Authors' Cooperation With Dr. Pizka 28
3.6 RTL Dumps From the Example Code 28
3.7 Scheme Users Benefiting From the Project 29
3.8 Stirring the Pot 29
3.9 The Solution Provided By Cygnus 30
3.10 Chapter Summary 31

4 Evaluation 33
4.1 Introduction 33
4.2 The Cygnus Solution to the Tail Call Problem 33
4.3 Mr. Jelinek's Sparc Patch 34
4.4 Unforeseen Problems Caused by the Patches 35
4.5 Testing the Patches on Ericsson's Test Program(s) 36
4.5.1 The Function Call Test Program 36
4.5.2 The Void Function Call Test Program 38
4.5.3 The Function Pointer Test Program 39
4.6 Chapter Summary 40

5 Conclusion 43
5.1 Introduction 43
5.2 Background Reading for the Project 43
5.3 The Timetable and the Project Phases 43
5.4 Writing the Report 45
5.5 The Dynamic Evolution of GCC 45
5.6 Experiences Gained 47
5.7 What Was Achieved 49
5.7.1 Non-Ericsson Specific Benefits 50
5.7.2 Benefits to Ericsson 50
5.8 What Could Have Been Done Differently? 51
5.9 Recommendations for Further Investigations in This Area 52
5.10 Final Words 56

References 57
Personal Contacts 57
Books and Papers 57
URLs 58

A Appendix. Influence of Optimization Grades on RTL Code 59
A.1 Without Any Optimization 59

A.2 With Optimization.....62

B Appendix. The Cygnus Patch.....65

B.1 The Added File sibcall.c.....65

C Appendix. A GCC Patch.....73

C.1 Mr Jelinek's Sparc Patch.....74

D Appendix. Concepts.....87

List of Figures

Figure 2.1 Threaded interpreter [HOG00] ... 6
Figure 2.2 Phases of a compiler ... 8
Figure 2.3 Simple syntax tree ... 9
Figure 2.4 Translation of a statement [AHOS86] ... 12
Figure 3.1 The function call test program ... 20
Figure 3.2 The function pointer test program ... 23
Figure 3.3 The gic compiler ... 27
Figure 4.1 The function call test program ... 37
Figure 4.2 The void function call test program ... 39
Figure 4.3 The function pointer test program ... 40
Figure 5.1 Timetable ... 44
Figure 5.2 Actors influencing GCC evolution ... 46
Figure 5.3 The GCC tree ... 46

1 Introduction

1.1 Defining the Problem

This project is initiated by Ericsson Infotech, Department of Test, Support and Simulated Platforms (TSP) in Karlstad. At TSP simulation of telephone switches is performed. When simulating, gotos are used in the simulation language (C). Newer versions of the C compiler, GCC, do not support the use of gotos between functions. Ericsson needs to develop support for a technique called "tail calls" in order to replace the code using gotos. This project involves investigating the requirements for GCC to support tail call elimination. As a background, the authors will describe in general terms the function and phases of a compiler, and in particular those of GCC.

The background and assignment as described by Ericsson. [PC7] will be presented in the following sections.

1.1.1 Background:

"GCC[1] is a free C/C++ compiler available for most commonly used computing platforms including Linux/i386, Windows/i386 and Solaris/Sparc. The source to GCC is freely available to anyone. At Ericsson Infotech we use GCC for most of our development work. In one of our simulation products we have code where every function ends with a call to a function with the same signature[2]. In order to speed up the code and avoid running out of stack space we use a special extension in GCC that allows us to use goto to jump to a label in another function. A better solution is to have a compiler that performs tail call elimination. When a function ends with a call to another function with the same signature, we know that we can reuse the space in the run-time stack for passing arguments to the function and avoid most of the function call overhead."

---

[1] GCC is an abbreviation for "GNU Compiler Collection". Earlier the abbreviation stood for "GNU C Compiler".

[2] Two functions have the same signature when they have the same type and number of arguments.

1.1.2 Assignment:

"To investigate if it is possible to implement tail call elimination in the current release of GCC (2.95). The goal is to implement the optimization for the Sun Sparc/Ultra Sparc architectures, but if a generic implementation is possible, that is preferred."

1.2 Scope of the Assignment

After interpreting the background and assignment that was given to us by Ericsson, we have, together with Ericsson reached a conclusion. This conclusion is that this project will be divided into one or possibly two parts.

The first part of the project involves a certain amount of research in order to find out if it is possible to implement tail call elimination in GCC 2.95. Since tail recursive calls already are implemented in GCC and the background material from Ericsson describes calls with the same signature, we can definitely say that the scope of the project in the tail call area has been narrowed down to sibling calls. When it comes to choise of hardware, the primary target enviroment is the Sun Sparc/Ultra Sparc architecture, but if possible a generic hardware solution is prefered.

The second part of the project is dependent of the outcome of the first part. If we in the first part reach the conclusion that an implementation of tail call elimination is possible, an attempt to create such an implementation will be performed.

1.3 Reading This Report

Chapter two gives the background to, and an overview of the project. The reader will be introduced to the project's initiator, Ericsson Infotech AB. At Ericsson Infotech in Karlstad, Department of Test, Support and Simulated Platforms, simulation of telephone switches is performed. When simulating a telephone switch CPU, gotos are used. The problem is that the new version of GCC does not support the use of gotos between functions. Ericsson needs to develop support for tail calls in order to replace the code using gotos. A tail call is when a function ends with a call to another function. GCC does not currently support a mechanism for handling tail calls. The reader will have an opportunity to acquaint himself with Compilers, and especially GCC. The chapter will, in general terms, describe the function and phases of a compiler.

The GNU Compiler Collection (GCC) is an open source compiler system. GCC's frontend supports several programming languages. The backend of GCC generates a machine

dependent, Lisp-like internal code before it optimizes the code and generates machine code. When compiling a source code in GCC, one can choose the level of optimization by setting a flag.

Another topic covered in this chapter is the platform Ericsson uses when the tail call problem occurs in their simulation

Chapter three describes how we will commence the task by studying various sources, i.e. papers, books and the Internet, in order to obtain adequate knowledge of the problem. By subscribing to mailing lists, we found knowledgeable persons. These persons have been a great source of information when it comes to our understanding of the problem, and also for the development of a solution to the tail call problem.

We will, with the help of code examples expose the problem with tail calls and also why the use of gotos in the newer versions of GCC causes problems that did not occur in older versions. If the tail call problem were to be solved, Ericsson would be able to upgrade their GCC software.

Several other related projects, which in different ways would benefit from a solution to the tail call problem, have been found. At the School of Computer Science in Toronto, the goal is to introduce Continuation Passing Style (CPS) for C programmers.

People from other projects have been involved in mailing list discussions of possible solutions to the tail call problem. Jakub Jelinek and Jeffrey Law have both been involved in creating implementations of those suggestions.

The authors came in contact-, and cooperated, with a Dr. Pizka in order to find a solution to the problem. Dr. Pizka is involved in the development of compilers which use the GCC backend, i.e. the gic- and C-- compiler.

Optimization in RTL by obtaining RTL dumps will be evaluated. By comparing dumps from different source codes and different stages of the compilation, one can better understand the connection between source, intermediate, and target code.

From the point we joined the mailing lists, the activity and interest for the tail call problem increased. A solution to the tail call problem was released by the Cygnus Company, and our project changed direction.

Chapter four evaluates the provided solution. The Cygnus patch is the major part of the solution, since it is this patch that deals with evaluating whether the function call is a tail call or not. This patch supports several target architectures, but not Sparc. One week after the Cygnus patch was released; Jacub Jelinek provided a patch with a solution for Sparc.

The Cygnus/Jelinek solution to the tail call problem is very detailed. Our experience is insufficient to evaluate the full details of this solution. However, at the same time, the people involved in the solution are experienced within the area of compiler development and GCC. They are hence able to provide a reliable solution to the tail call problem.

As always, consequenses of changes in GCC are difficult to predict. The tail call solution caused a variety of minor negative side effects that will be presented.

The authors will use Ericsson's test program in order to verify the patches suitability for Ericsson's purposes. The outcome of the different test programs will be presented and the reader will receive the answer to whether the Cygnus/Jelinek solution has brought Ericsson to a final solution, or if the problem remains.

Chapter five will describe the achievements gained in this project. The issues of background reading and how the timetable was used in order to optimize each phase of the project will also be shown.

The process of development of GCC will be presented as well as the experiences the authors have gained during this project. The most important issue of the project will be described in the final sections of this chapter.

The reference section contains references to books, papers, Internet URLs and also personal contacts.

2 Background

2.1 Introduction

This chapter will explain the background to the project. The project's initiator Ericsson Infotech AB will be introduced to the reader. The authors will also try to give the reader an overview of the project. We will discuss the ideas behind the project, the compiler, and especially GCC. Other topics in this chapter will include optimizing and the platform Ericsson uses when the problem occurs in their simulation.

2.2 Ericsson, As the Company Describe Itself

This is a project, which has been initiated by the Department of Test Support and Simulated Platforms at Ericsson Infotech AB in Karlstad. Infotech is a part of the international Ericsson group. Ericsson is a world-leading supplier in the fast-growing and dynamic telecommunications and data communications industry, offering advanced communications solutions for mobile and fixed networks, as well as consumer products. Ericsson is a total solutions supplier for all customer segments: network operators and service providers, enterprises and consumers. Ericsson has more than 100,000 employees, representation in 140 countries and clearly the world's largest customer base in the telecommunications field [URL11]. The authors have chosen not to challenge these statements. Instead we leave that task to the supporters of Ericsson's competitors. The presentation of the company does not claim to be, and is not intended to give an objective picture of the company. It is merely a generic orientation for the reader.

Ericsson Infotech AB with over 550 employees is a product and development company in the field of mobile telecommunications, located in Karlstad. They have product and development responsibility within a number of product areas, including Signaling System No.7 and protocol converters, APZ emulators and simulators, wireless Internet solutions, radio net products, as well as maintenance and customer support systems [URL12].

The Department of Test Support and Simulated Platforms (TSP), is a department at Ericsson Infotech. The department's goal is to become Ericsson's leading supplier of simulator products as regards platforms, systems, and network solutions. The mission is to

offer products and services - based on APZ Emulators and Simulators - for Ericsson's customers to improve business and within Ericsson to reduce costs [URL13].

2.3 Project Overview

When simulating a telephone switch, that is a combination of software and hardware, software is used in order to simulate the hardware. Each instruction in a simulated telephone switch CPU is represented by a procedure call. Every procedure ends with a call to another procedure. This call sequence is named tail calls. For each call, information about parameters and status is saved to the program stack. Eventually, the stack will run out of space and the application crash with a segmentation fault.

Today this problem has been temporarily solved by the use of an extension to the GNU C Compiler that allows gotos to jump to labels in other procedures. A goto instruction does not consume stack space, as an ordinary parameter passing function call would. This will prevent the stack from growing with each call.

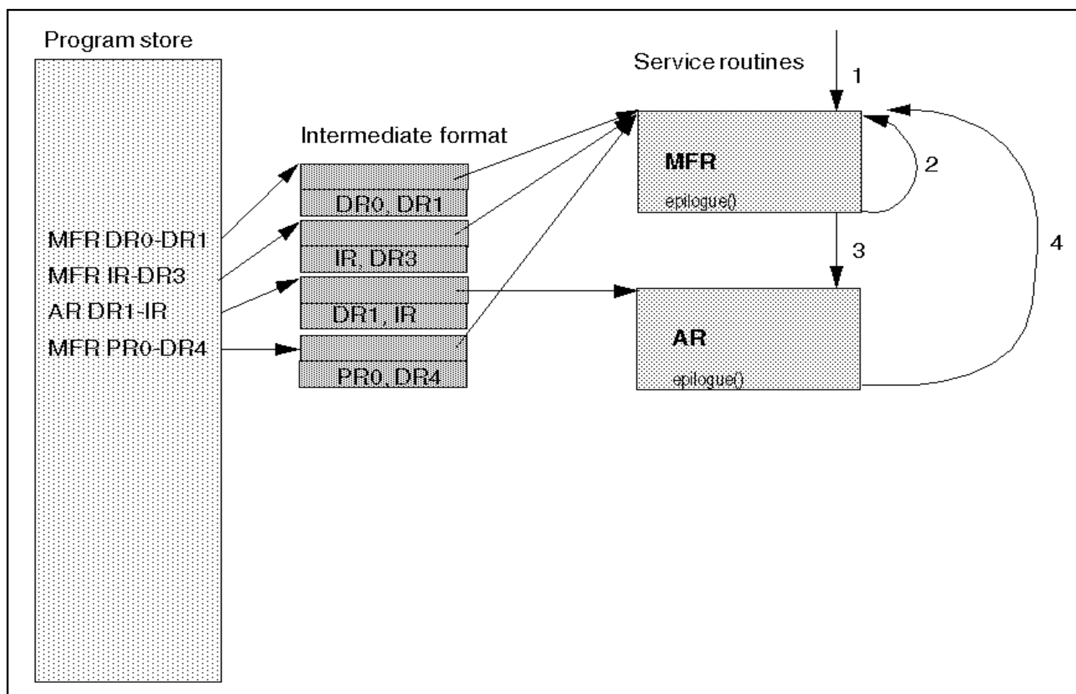


Figure 2.1 Threaded interpreter [HOG00]

Figure 2.1 gives an overview of how TSP's APZ simulator works. Intermediate code is generated during runtime, as soon as an instruction is to be executed that has not been decoded before this is performed. Decoding is performed one basic block at a time until first branch instruction. Intermediate format for each instruction is 64 bits, 32 bits service routine pointer and 32 bits of instruction parameters. If parameters do not fit in 32 bits (few instructions), the 32 bits instead becomes a pointer to allocated structure where parameters are filled and retrieved. Parameters are packed for best possible performance when fetching them.

The source code of the simulated CPU is generated automatically with a meta-tool, *SimGen*. Since tail call elimination is not supported, the code generator is tuned to generate code with gotos. If it is possible to eliminate tail calls, the code generator can be tuned to produce code without gotos.

Compilers and compiler construction is a highly complex subject which requires insight into a variety of areas. For GCC one should be able to fully understand the GCC terminology, e.g. RTL, tree structure, passes and flow analysis. In order to obtain the necessary knowledge, the authors of this report have read several publications in the subject of designing and implementing optimizing compilers [AHOS86, LUN91, MAS99, MUC97, PIZ97, PIZ00, STA99, WES92, URL5, URL7].

In the remainder of this chapter we will introduce some of the most important aspects of this project and also explain these in further detail. This explanation is presented in order to give the reader a wider perspective and a suitable starting point in order to be able to understand the problem and the complexity of the solution.

2.4 Tail Calls

A tail call is when a function ends with a function call. If the callee is the same function as the caller, this is called a self-recursive call. Another kind of tail call is the sibling call. When a function calls another function and they both have the same signature, i.e. the same number and type of arguments, it is named sibling call. An example of a sibling call will be shown in Figure 3.1.

2.5 The Functionality of a Compiler in General

In order to help the less experienced reader to better understand this report, we will try to explain the functionality of a compiler. A compiler works in phases [AHOS86]. Each phase

transforms a source program to a different representation. A typical picture of a compiler is shown in Figure 2.2 [AHOS86].

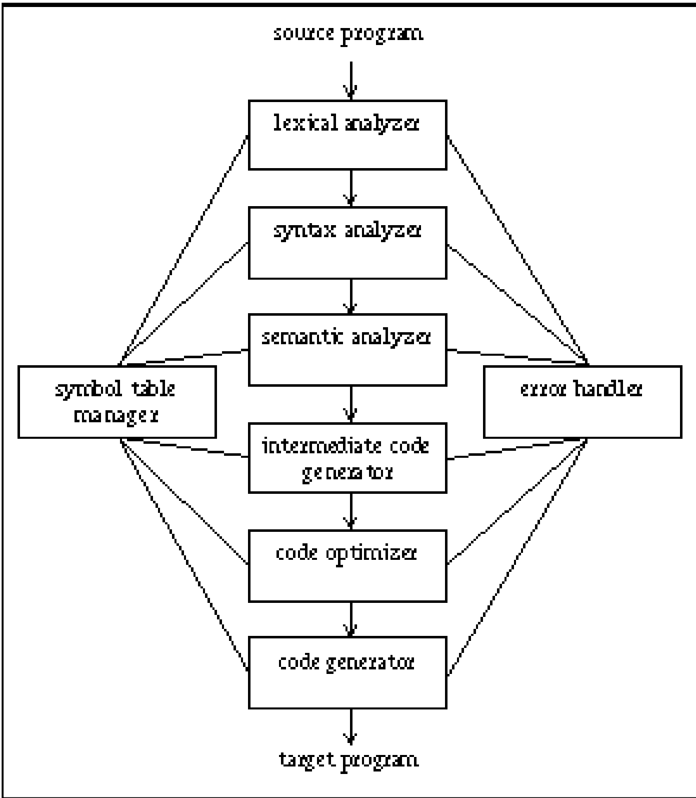


Figure 2.2 Phases of a compiler

The compiler is divided into a frontend and a backend. The frontend is where the source language is transformed from a high level language into a machine independent intermediate representation. The frontend encompasses lexical analyzer, syntax analyzer, semantic analyzer, creation of a symbol table and generation of intermediate code. The frontend also includes error handling for these phases. The backend is where the machine- and intermediate language dependent phases belong. It is in the backend where most of the code optimization is performed and where the machine code is generated. As in the frontend, necessary error handling is also included. The phases described in sections 2.5.3 to 2.5.6 will be further explained in an example (Figure 2.4). The connection between the phases of a general compiler and GCC's phases will be described in section 3.4.5.

2.5.1 The Symbol Table Manager

The Symbol Table is a data structure, which holds information about identifiers such as functions and variables. The information could be about type, scope and number of arguments. During compilation the symbol table manager is called in order to control the validity of an identifier.

2.5.2 Error Handler

Each phase can find errors that must be handled in some way in order to be able to continue the compilation without halting at each error. Every error is reported to the error handler that handles the error in an appropriate way. When one wants to use a debugger the detected errors stored in the error handler is very helpful.

2.5.3 The Analysis Phases

First the lexical analyze phase reads the characters in the source program. While doing so, the lexical analyzer eliminates all spaces and return keystrokes in the code. This procedure is performed in order to produce token streams. A token could be an identifier, a keyword, a punctuation character or a multi-character like '!='.

The second phase of the analysis phase is the syntax analyzer. The syntax analyzer checks if the code follows the grammar rules of that specific programming language. E.g. a function name must be followed by an open bracket, '('. While performing such tests, the syntax analyzer creates a syntax tree. An example of a simple syntax tree for the expression "(9-5)+2" is shown in Figure 2.3.

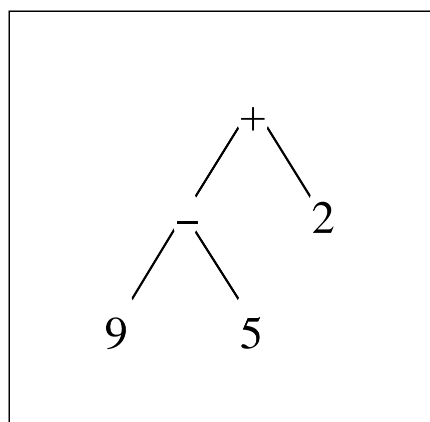


Figure 2.3 Simple syntax tree

The third and last phase of the analysis is the semantic analysis where the compiler examines the code to see if it follows the semantic conventions of the source language. E.g. when the same variable is declared twice in a function, both declarations follow the syntactic conventions but by declaring the variable two times, a violation of the semantic convention is performed.

2.5.4 Generation of Intermediate Code

Some compilers produce an explicit intermediate representation of the source program. There are two important properties that the intermediate code should have. The code should be easy to produce and easy to translate into the target program.

2.5.5 Optimization

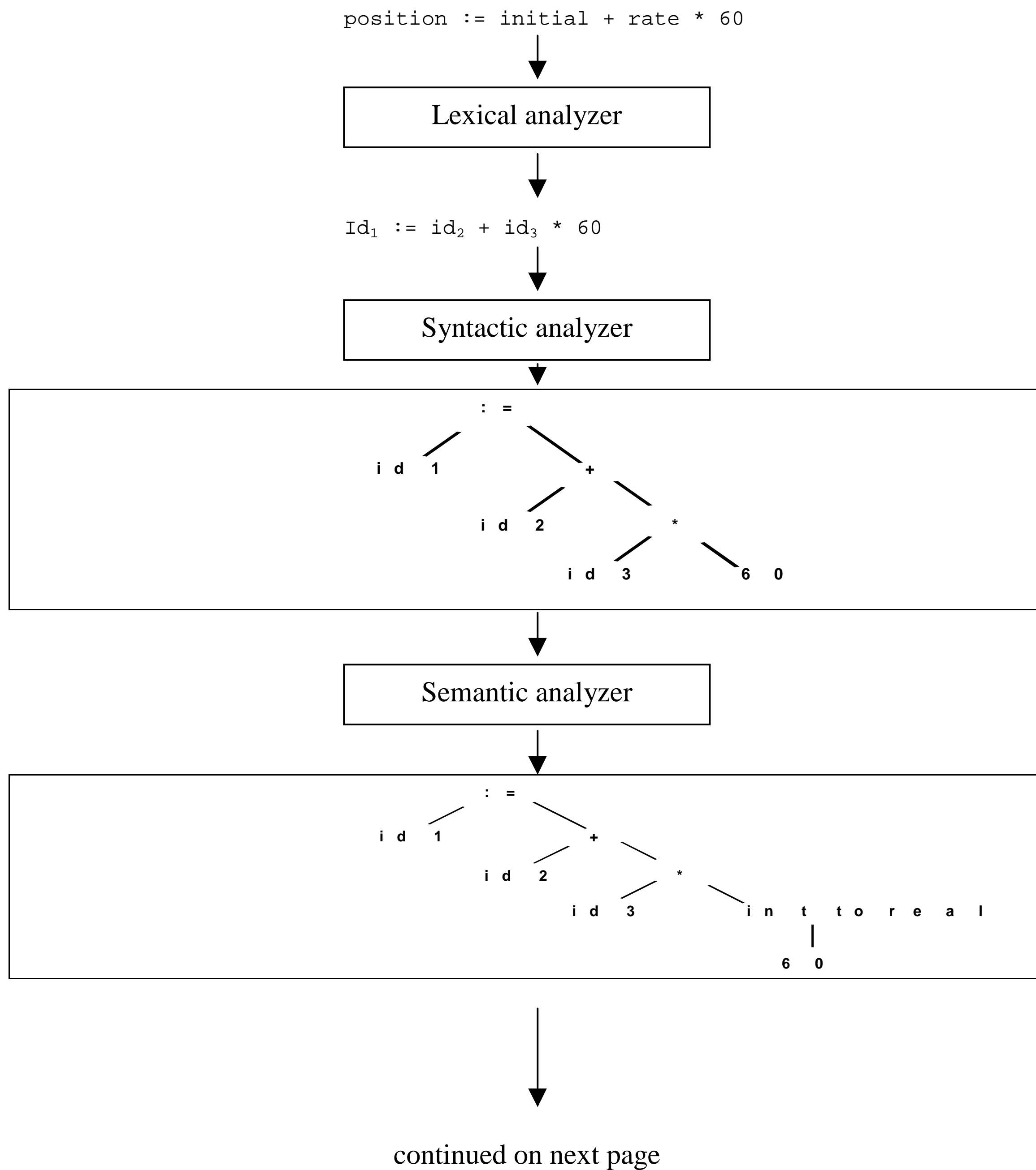
The code optimizer's task is to improve the intermediate code in order to produce faster running machine code. Also, the order of instructions can be altered to render the code more efficient and to make better use of the registers and the instruction set for the CPU.

2.5.6 Generation of Target Code

The final phase of the compilation is the generation of assembler code or target code. Intermediate instructions are translated into a sequence of machine code. Memory locations are selected for the variables. The assignment of variables to registers is a very important issue since one has to be convinced that the register is not already used for e.g. another variable at the same time.

2.5.7 Example of Translation of a Statement

Let us take a look at an example code. We will in Figure 2.4 translate a Pascal statement.



continued on next page

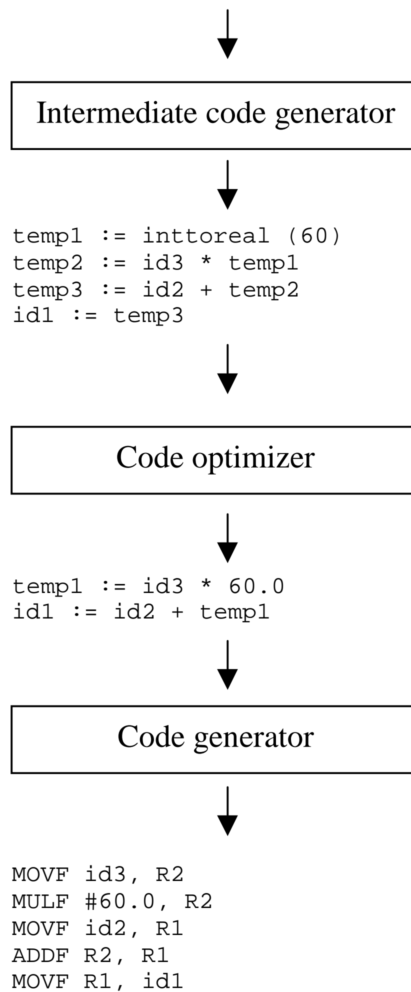


Figure 2.4 Translation of a statement [AHOS86]

As Figure 2.4 shows, the source code is translated step by step during the different phases of the compilation as described in sections 2.5.3 to 2.5.6.

2.6 The Compiler System GCC

At Ericsson's Department of Test support and Simulated Platforms, GCC is used for most of the development work. GCC, an abbreviation for GNU Compiler Collection (former Gnu C Compiler), is continuously developed and maintained by the Free Software Foundation, and distributed under the GNU General Public License (GPL). GNU stands for "Gnu's Not Unix!" According to FOLDOC, a hackish tradition is to choose acronyms and abbreviations that refer humorously to themselves [URL6]. GNU software is always distributed with its

sources and anyone may modify the software and redistribute the modified code, provided that the modified sources are made available in turn. If the modified source code follows the guidelines of the official FSF development tree[3], the code can become a patch[4] and eventually used in a later release of GCC.

GCC is the centerpiece of the GNU software. GCC is a compiler system for multiple source languages such as C, C++, Objective C, Fortran77 and Java, which produces code for such hardware targets as Linux/i386, Windows/i386 and Solaris/Sparc. GCC consists of a frontend, an intermediate representation and a backend. [STA99]

The front-end transforms the high level language into a machine independent intermediate representation, referred to as an abstract syntax tree. The front-end usually encompasses scanning, parsing, semantic analyzes and finishes with the synthesis of GCC trees [PIZ97]. In the backend the abstract syntax tree is converted into Register Transfer Language (RTL), which is a Lisp-like, machine dependent internal representation. The tree and RTL is called the intermediate representation of the code. Most of the work of the compiler is performed on the RTL code. In order to use the registers in the best possible way and to increase the speed of the executable file, optimization is performed on RTL. The connection between the phases of a general compiler and GCC's phases will be described in section 3.4.5.

To better appreciate the complexity of the task, let us take a look at GCC in a wider perspective. The compiler consists of a total source approaching one million lines of code [MAS99]. More than 1700 files in about 60 different directories share these lines of code.

2.6.1 The Passes of GCC

When GCC compiles source code into machine code, the compiler does this in a number of different passes, which involves several files of the compiler. The RTL intermediate code for a function is generated as the function is parsed, a statement at a time. Each statement is read in as a syntax tree and then converted into RTL, then the storage for the tree for the statement is reclaimed. Storage for types (and the expressions for their size), declarations and a representation of the binding contours and how they nest, remains until the function is finished being compiled. This information is needed for debugging purposes. The overall control structure of GCC is in the file 'toplev.c'. This file is responsible for initialization, decoding arguments, opening and closing files and sequencing the passes. Each pass involves

---

[3] We will in future chapeters refer to the FSF development tree as the GCC tree.

[4] A patch is piece of code that is inserted in an existing file. Appendix C will give an example of a GCC patch and we will also describe the patch procedure.

one or more different files. In the list below, the passes of GCC, will be briefly described in their usual sequence [STA99].

- Parsing. This pass reads the entire text of a function definition and constructs partial syntax trees.
- RTL generation. Converts syntax trees to RTL code. Optimization for 'if'-conditions that are comparisons, Boolean operations or conditional expressions are evaluated. Tail recursion is also detected here. At the end of this pass decisions are made if the function can and should be expand inline[5] in its caller.
- Jump optimization. This pass simplifies jumps to the following instruction. It also deletes unreferenced labels and unreachable code.
- Register scan. Finds the first and last use of each register.
- Common sub-expression elimination. This pass also performs propagation of constants.
- Global common sub-expression elimination. Also performs global constant and copy propagation.
- Loop optimization. Moves constant expressions out of loops and optionally does strength reduction and loop unrolling as well.
- Data flow analysis. This pass divides the program into basic blocks and deletes unreachable loops. Then it computes which pseudo registers are live at each point in the program, and makes the first instruction that uses a value point at the instruction that computed the value. The pass also deletes computations whose results are never used, and combines memory references with add or subtract instructions to make auto increment or auto decrement addressing.
- Instruction combination. The pass tries to combine groups of instructions that are related by data flow to single instructions.
- Register movement. The pass looks for cases where matching constraints forces an instruction to need a reload and this reload would be a register-to-register move. The pass then attempts to change the registers used by the instruction to avoid the move instruction.

[5] One can say that inline expansion is when a function's code is placed directly into its caller to avoid an expensive function call

- Instruction scheduling. This pass looks for instructions whose output will not be available by the time they are used in subsequent instructions. The pass also reorders instructions within a basic block to try to separate the definition and the use of items that otherwise would cause pipeline stalls.
- Register class preferencing. The RTL code is scanned to find out which register class is best for each pseudo register.
- Local register allocation. Allocates hard registers to pseudo registers, which are used only within one basic block.
- Global register allocation. The pass allocates hard registers for the remaining pseudo registers.
- Reloading. Renumbers pseudo registers with the hardware registers numbers they were allocated. Pseudo registers that were not assigned hard registers are replaced with stack slots.
- Instruction scheduling is repeated.
- Jump optimization is repeated.
- Delayed branch scheduling. An optional pass that attempts to find instructions that can go into the delay slots of other instructions, usually jumps and calls.
- Branch shortening. On most RISC machines, branch instructions have a limited range. Therefore longer sequences of instructions must be used for long branches. In this pass the compiler figures out how far each instruction will be from each other instruction, and hence if the usual instructions, or the longer sequences, should be used for each branch.
- Final. Outputs the assembler code for the function. This pass is also responsible for identifying spurious test and compare instructions. Machine specific peephole optimizations are performed at the same time. It is also here that the function entry and exit sequences are generated. These sequences never exist as RTL.

2.7 Optimization Options in GCC

When compiling source code, one can choose different levels of optimization by including and setting flags in the command row as shown below.

```
gcc test.c –O1 –o outputfile
```

In short one can describe the different levels of optimizing as follows: [STA99]

- '-O' Sets a number of other flags in order to enable calls to functions, which performs different types of optimizations. This flag is always used as 'O0', 'O1', 'O2', 'O3' or 'Os'.
- '-O0' (capitol O-zero) gives the compiler the instruction "Do not perform **any** optimization."
- '-O1' The compiler tries to reduce code size and execution time.
- '-O2' Optimize even more. The compiler performs nearly all optimization that does not involve a space-speed trade-off. GCC does not perform any loop unrolling or function inlining. Increases the compilation time as well as the performance of the generated code
- '-O3' Same as '-O2' but supporting inlining as well.
- '-Os' Optimize for size only. Uses all '-O2' functions that do not increase code size and also other specially designed functions in order to reduce code size.

Any use of the optimization flags will cause an increasing compilation time and memory usage.

2.8 Platform

At Ericsson's Department of Test support and Simulated Platforms, the platform on which the software is run is the Sparc32 architecture. Hence our project will only concentrate on solving the tail call problem for the Sparc32 architecture. Sparc stands for "Scalable Processor ARChitecture" and was originally designed by Sun Microsystems in 1985. Sparc has been implemented in processors used in a range of computers from laptops to supercomputers.

2.9 Chapter Summary

This chapter has brought up the issues of this project's background. The Department of Test, Support and Simulated Platforms at Ericsson Infotech AB in Karlstad initiated the project. Ericsson is a worldwide supplier of infrastructure for telecommunication and data communication. At Ericsson Infotech in Karlstad, simulation of telephone switches is performed.

When simulating a telephone switch CPU, gotos are used. Newer versions of GCC do not support the use of gotos between functions. Ericsson needs to develop support for tail calls, which is when a function ends with a call to another function, in order to replace the code using gotos. The architecture on which simulation is performed is Sparc.

The chapter has in general terms described the function and phases of a compiler. First, the compiler analyzes the source code and transforms it into an intermediate representation, optimization is performed and finally machine code is generated.

The GNU Compiler Collection (GCC) is a free open source compiler system. GCC's frontend supports several high level programming languages. The backend of GCC generates a machine dependent Lisp-like internal representation, RTL. The RTL code is optimized and machine code is generated. Between the frontend and the backend, an intermediate representation in form of a abstract syntax tree is used.

When compiling a source code in GCC, one can choose different levels of optimization by setting a flag.

3 The Task

3.1 Introduction

In this chapter we will elucidate the issues of how we begun the work on this project. The problem that occurs in Ericsson's simulation program will be thoroughly reviewed. We will also present some related projects and how the authors came in contact with, and worked together with a person at Microsoft Research. This project's effect on the surrounding environment and how this effect might have lead to a solution to the tail call problem will be illustrated too.

3.2 Beginning the Task

In order to obtain adequate awareness of the problem, the authors have studied a variety of different sources as mentioned earlier. The first measure was to read chapters one to three of "The Dragon Book" [AHOS86]. This book gave a good outline of how a compiler is constructed and how it works in general. Another source of information we have used is the GCC mailing lists [URL2, URL3], which is an excellent way to obtain information. One can read or search for questions that were asked during the last two years by subject or date. The lists that we have searched and also subscribed to are the "gcc" list and the "gcc-patches" list. The "gcc" list deals with questions of a common nature while the "gcc-patches" list handles the issues of how to improve GCC by adding patches.

Our supervisor at Ericsson, Magnus Einarsson posted a question about tail calls in October 1999 to the "gcc" list. Within 24 hours he received nine answers. These answers became a starting point for our further studies. The reason we can make this statement is that these mails provided us with names to experienced people, ideas how to solve the problem of tail calls and also a solution created by Jacub Jelinek[6].

---

[6] Jacub Jelinek who is an employee at the Red Hat Company will be further presented in section 3.4.2.

3.3 The Problem

The main motivation for this project will be discussed in the following section. We will first show an example of source code, written in C that will produce the same type of problem as the real simulation program problem. Figure 3.1.

```
int sibling1(int x1);
int sibling2(int x2);

int main()
{
      int x0 = 1;
      sibling1(x0);
}
int sibling1(int x1)
{
       printf("%i\n",x1);
       x1 = x1+1;
       return sibling2(x1);
}
int sibling2(int x2)
{
      printf("%i\n",x2);
      x2=x2+1;
      return sibling1(x2);
```

Figure 3.1 The function call test program

The main function is the entry point for the program and initializes the variable x0 to zero. In main the function sibling1 is called with actual parameter x0. In sibling1, the value of x0 is bound to x1. The value of x1 is displayed on the screen. The parameter value is displayed in order to keep track of the call sequence leading to the segmentation fault. Every time an odd number is displayed, it shows that sibling1 is executing. Correspondingly, an even number is displayed when executing sibling2.

The display will eventually show how many function calls have been made before the stack runs out of space. After writing the parameter value to the screen, x1 is incremented by one. The function sibling2 is called in a similar manner and returns x2 to sibling1 again. Now that the call sequence has started, we have the same situation as in the test program run by Ericsson's test department. This is where the problem starts. For each call, the return address and the value of the parameter are pushed on the program stack. Eventually, the stack will run out of space and the application crash with a segmentation fault. As mentioned earlier, this problem has been temporarily solved by the use of a GCC extension that allows gotos to jump to labels in other functions. As mentioned in section 2.3, a goto instruction will not consume stack space and therefore the problem of a full stack never occurs. A program using gotos is illustrated in Figure 3.2 below.

Ever since Edsger W. Dijkstra's article in the March 1968 "Communications of the ACM", "Goto Statement Considered Harmful", the usage of gotos has been discussed within the programming community. Nowadays most people involved in programming projects agree on the fact that the use of gotos may lead to unstructured programs, which in turn makes the code difficult to read and understand. It is therefore highly recommended that gotos should not be used.

In version 2.8.1 of GCC, which is currently used at Ericsson, the use of gotos works fine, but is not recommended. In the GCC manual for version 2.8.1 one can read; the use of gotos is considered undefined behavior, but **can** be used for jumps between functions. In later versions of GCC the text in the manual has changed to; gotos is considered undefined behavior and **can not** be used for jumps between functions. The simulation program using gotos will hence not be able to run under the newer versions. The problem of not being able to run the simulation program could be solved by not updating GCC. Of course, not updating GCC in particular or a company's software in general, is a solution that will not work in the long term. There could be several problems involved in using non-updated software. One is the risk of not being able to get support. There is also a risk of trouble when cooperating with others working with updated versions of a program.

The reason for the change in the manual is that later versions of GCC performs optimization "function at once", rather than "statement at once" as done in GCC, version 2.8.1. "Statement at once" evaluates the code statement by statement in a narrow approach, not taking the wider view into consideration. Below, in the corresponding text to Figure 3.2 we will give a more in depth explanation of the two approaches presented here.

“Function at once” evaluates the complete function, rather than statements and **does** take the wider view into consideration. The benefit of “Function at once” is that it enables a higher grade of optimization. The higher grade of optimization however, makes it impossible for Ericsson to use gotos in their simulation. The reason why Ericsson cannot use gotos in their simulation is that when the compiler performs optimization at RTL level, the compiler tries to use the fast registers in the best possible way. The compiler does this by changing the order of code while initializing the variables in order to avoid unnecessary swapping. If the compiler finds dead code, i.e. code that is considered never to be used, the compiler simply disregards that code and hence do not generate assembler code. The problem with gotos and the better optimization in “function at once“ will be illuminated in the text corresponding to the example shown in Figure 3.2. Unfortunately this code example was not presented to us until April 14th. This is the code that corresponds closest to the real simulation program.

```
01 int x = 0;
02
03 void one();
04 void two();
05
06 void *array[2];
07
08 int main()
09 {
10       one():
11       two();
12       goto *array[0];
13 }
14 void one()
15 {
16       array[0] = &&onelabel;
17       return;
18       onelabel:
19       printf("One %i\n",x++);
20       goto *array[1];
21 }
22  void two()
23 {
24       array[1] = &&twolabel;
25       return;
26       twolabel:
27       printf("Two %i\n",x++);
28       goto *array[0];
29 }
```

Figure 3.2 The function pointer test program

The reason for this approach in programming style is to boost the performance of the simulator. Function calls will be costly in time due to the parameter passing, hence one wants to avoid function calls as much as possible. To avoid function calls this example code uses a global array (row 06) for storing pointers to labels. The addresses to the labels are collected and stored when main calls function one and two (rows 10-11). The only action function one and two performs when called, is to copy the label addresses into the global array. (rows 16

and 24). They then return to the caller (rows 17 and 25). After this procedure all other interfunction communication is made by jumps to labels inside functions. E.g. from row 28 a jump is made to row 18.

In GCC version 2.8.1 the program behaves as described above while the later versions of GCC will behave as shown as in the following example: Let us take a closer look at the function called one (row 14 to 21). When the compiler evaluates the function, it checks the code to see if there is any unreachable code. The compiler notices that a jump to the label at row 18 never occurs within the code in function one. After detecting the supposed dead code at rows 18 to 20, the compiler disregards them when creating the assembler code. The compiler also tries to optimize the register handling when it initializes the variables by changing the order of the variable depending on in which order the variables will be used. The conclusion of this example is that the use of gotos and labels will cause perfectly good code to be ignored and dismissed as unreachable.

3.4 Related Projects

This section will describe the different researchers that have been found on the Internet. Our supervisor, Mr. Einarsson, found one of the researchers, the School of Computer Science, as he was performing benchmarking. The goal was to get some background information about this project and also evaluate the value of such a project. The other actors were found on the GCC mailing list.

3.4.1 The School of Computer Science in Toronto

At the School of Computer Science in Toronto, Canada, a group has investigated the possibilities of a modification to GCC that would support tail call elimination. The primary motivation for their work was to enable Continuation Passing Style (CPS) in C since CPS resolves a longstanding structural problem in the composition of software reliability [MAS99]. Experiments with several different program styles were conducted. Most functions (fragments) that result from CPS conversion will want to pass control to a different function. This is where the CPS tail call problem occurs.

The interest of their work lies far from ours but there is one common denominator: tail call elimination. When the group started to look into the possibilities of invoking generic tail call elimination in GCC they discovered that it was quite difficult. As they write: “A very complex 1600 line function would have to be rewritten to support general tail call removal. Not only is the function large, but also intimately interwoven with the rest of the compiler.

With the total source approaching a million lines of code, even if only ten per cent is involved, this remains a daunting task!” In their paper they mention that a lesser solution would get most of the gain by modifying the existing self-tail recursion code.

By March 1999 the group’s work was still ongoing but since then they have abandoned that project due to lack of time and other resources.

3.4.2 Jacub Jelinek

Mr. Jelinek is one of the most active people on the gcc-patches list when it comes to mail correspondence and the production of patches. He is an employee of the Red Hat Company where he works with the Linux operating system and especially on the Sparc architecture. In the correspondence in which Mr. Einarsson, our supervisor was involved in, in October, Mr. Jelinek claimed to have a solution to the tail call problem for the Sparc architecture. The accuracy of Mr. Jelinek’s patch was debated and deemed not to be the “right” solution. The reason for this judgment was that the tail call check must be handled earlier in the compiler than Mr. Jelinek’s solution did. We have not been able to test and evaluate the patch in our project since the patch is constructed for an older version of GCC (2.7), which is not available for download from GCC’s homepage anymore. The GCC version the patch was created for also belongs to an older generation of GCC. With the 1999 GCC version 2.95, a new generation of GCC was born, hence a comparison between the versions is quite inappropriate.

Mr. Jelinek would later provide the GCC patches list with a patch, which would be the link to a complete solution for Sparc. See Appendix C.1.

3.4.3 Marc Lehman and Per Bothner

Mr. Lehman and Mr. Bothner both took part in the debate concerning Mr. Jelinek’s patch. Their opinion on the subject was as follows: Mr. Bothner, who is an independent consultant, claims that detecting that something is a tail call should be performed at the tree level. However, the actual hard part of the stack adjustment should be performed at a rather low level. A problem with tail calls in standard C calling conventions is that the caller pops the argument from the stack. It would be better if the callee pops the argument. Mr. Bothner feels that the fully general tail call case is hard to implement, but a sibling call with the same parameter type would be relatively simple to implement. “Just evaluate the arguments into their proper outgoing parameter locations, and jump to the target’s address.“ He also says that a sibling call solution would be valuable, as it would provide a cleaner solution for those people who want to compile functional languages such as Scheme and ML into C.

Mr. Lehman, an employee at the Technische Universität in Karlsruhe and also a member of the GCC steering committee, agreed to Mr. Bothner's ideas and added that the solution would also allow writing efficient threading interpreters without resorting to GCC extensions like computed gotos. Since TSP's simulation programs use computed gotos, efficient threading interpreters is just what they want![7] See Figure 3.2.

3.4.4 Jeffrey A. Law, the GCC Release Manager

Mr. Law was one of the most prominent participants of the Jelinek-patch-debate. As the release manager of GCC, he obviously has a greater insight in the structure and functionality of GCC. He also has access to the future plans of the GCC development and maintenance and therefore the ability to recognize a suitable solution. Mr. Law was the person who most objected to the functionality of Mr. Jelinek's patch.

Mr. Law agrees to what Mr. Bothner said but says that a general solution is not that hard to implement if the deal with "the arg stuff at the tree level (which is what my implementation does). It's not that bad, once you determine that there's enough space to use you're set." As a comment to the low level work, Mr. Law says that the stack, epilogue and related code needs to be done at the RTL level with RTL based epilogues and that it would be possible, but ugly to make them work with the old style epilogues.

In an answer to our supervisor Mr. Einarsson, Mr. Law wrote that he had created a solution to the problem of tail calls. The solution was created for MIPS, pa and x86 and was shipped to "selected Cygnus customers". His solution ports to targets with RTL epilogues easily by defining a sibling call epilogue pattern and sibling call patterns. According to Mr. Law, the code would need some cleaning up, particularly in its interface into the integration phase and exception handling. Writing the solution for MIPS, pa & x86, Mr. Law had the Sparc architecture in mind in order to simplify a future expansion. To the question if it was possible to obtain a copy of the code, Mr. Law chose not to answer. As the GCC Release Manager he received a large amount of emails every day and was certainly not able to respond to all. Mr. Law's solution to the tail call problem was later used as a starting point of another solution that will be presented in section 4.2.

[7] The authors comment.

3.4.5 PhD Markus Pizka at Microsoft Research in Cambridge, England

In February 2000, Dr. Pizka posted a contribution to the GCC mailing list, declaring his eagerness to find a solution to the tail call problem. Dr. Pizka had been working at the Technische Universität in München with the development of the GNU Insel-Compiler (GIC) among other duties. GIC uses the GCC back end and its own front end for the Insel programming language. Figure 3.3 [PIZ97] shows how GIC connects to the GCC backend, it also describes the phases of a compiler in general in a well-arranged way. One can say that the *static analysis* corresponds to the GCC frontend and the four first step shown in Figure 2.2. One can also say that GIC's *synthesis1* corresponds to the GCC intermediate representation and the output from the phase *intermediate code generator* in Figure 2.2. GIC's *synthesis2* corresponds to the GCC backend and the two final phases, *code optimizer* and *code generator* of Figure 2.2.

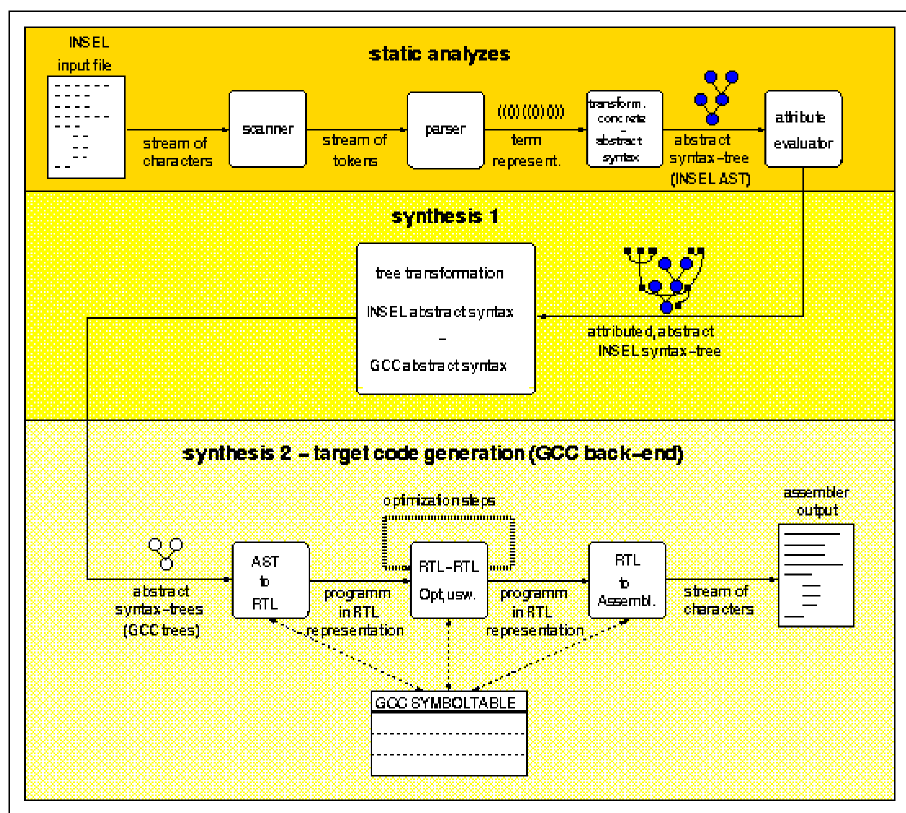


Figure 3.3 The gic compiler

Dr. Pizka is currently employed at Microsoft Research. One of his tasks at the moment is to investigate the C-- compiler. In both cases mentioned above, GIC and C--, GCC is a major issue and so is also the tail call problem. This is the reason why Dr. Pizka is so anxious to find a solution to the problem. We approached Dr. Pizka via the GCC mailing list, with some questions about tail calls and if he knew of any way to gain information or how to connect with people with knowledge in the area. The correspondence that followed almost instantly lead to the point where Dr. Pizka offered to support our effort by guiding us through the task of creating a solution to the tail call problem. According to Dr. Pizka himself, he had the capability but not the time to implement such a solution. The further cooperation with Dr. Pizka will be described in section 3.5.

3.5 The Authors' Cooperation With Dr. Pizka

First Dr. Pizka wanted to know more about our knowledge in the area of compiler construction, and also the schedule of our thesis. After receiving this information Dr. Pizka instructed us what we should read and where we could find the material for this purpose. He also informed us about which GCC files we should take a closer look at and try to understand. Having studied the recommended material we were able to ask questions of a more informed kind to Dr. Pizka. Dr. Pizka has kindly answered these questions and tried to give us a clearer picture of how a compiler works and how to solve the tail call problem. In the discussions with Dr. Pizka it became clear to us that RTL was an important part of the compiler and hence important to investigate further. We will in section 3.6 go further into details about RTL.

Our cooperation with Dr. Pizka stopped when Richard Henderson at Cygnus publicized the improved Jeffrey A. Law solution to the tail call problem[8]. After this point, as far as Dr. Pizka and we were concerned, there were no reasons for further investigations.

3.6 RTL Dumps From the Example Code

We have studied the GCC manual's chapter about passes and files of the compiler and RTL representation. While studying the manual, we discovered that it is possible to obtain dumps of the RTL code as the compiler performs the optimization. The benefit of such an RTL dump is the possibility to connect the GCC source code to the resulting RTL code as the outcome of our test program. We recognized the RTL dumps to be a help for us to better understand the

[8] This solution is presented in section 4.2.

optimizing stages of the GCC compiler. By understanding the RTL code and the connection between RTL, our test program and the GCC source code, we should be able to suggest and, if it fits our schedule, possibly implement the necessary changes to the GCC source code.

There are several different ways to obtain dumps of the RTL code as the compiler performs the optimization. By setting different flags before compiling a source file, one can choose how many steps the compiler will optimize before creating the dump. E.g. the option "-df" causes a RTL dump after the flow analysis pass. This is where the program divides the program into "basic blocks" and deletes unreachable code. This pass also controls which pseudo registers are live at each point in the program.

By using various options one can obtain RTL code in form of dumps from different stages of the compilation. The dumped RTL code is saved in a file. Appending an extension to the input file name automatically generates the file's name. Every option has a different extension. E.g. the "-df" option creates a file with the extension ".flow".

An example with different optimization levels from our evaluation of RTL code will be shown in Appendix A.

3.7 Scheme Users Benefiting From the Project

A possible side effect of the outcome of this project might be to satisfy Scheme users who want to convert Scheme into C-code and then to compile the code in GCC. There are many such Scheme-to-C compilers used today despite their complexity. The reason for converting Scheme to C is the large number of target environments supported by the GCC backend [PC6]. GCC does not support Scheme. Scheme is a program language that to a large extent uses recursion. Since GCC does not yet supports tail call elimination, the Scheme-to-C compiler not only converts Scheme to C, but also performs elimination of tail calls in order to increase the performance of the executable code.

If a solution to the tail call problem in GCC would emerge, it would enable the conversion from Scheme to C to be performed in a simpler manner. The Scheme-to-C compilers could therefore be designed in a less complex manner.

3.8 Stirring the Pot

From the point where we posted our first questions on the GCC mailing lists, the interest for tail calls has increased. The tail call discussion activity on these lists reached an "all time high" in the beginning of March. The discussion about Mr. Law's earlier solution led to an

agreement that an improvement of the earlier solution was the best way to go in order to solve the tail call problem. After about a week a suggestion to a solution was presented to the GCC patches list. The person who posted this suggestion was Richard Henderson, an employee of the Cygnus Company. Over a period of a week some improvement details were discussed and also implemented. A "final" version of the patch was implemented in a GCC snapshot (egcs-20000313).[9] Unfortunately the snapshot did not support the Sparc architecture.

From this point the authors project changed direction. After conferring with our advisor at Karlstad University, Donald F. Ross, we decided to make some changes in the aim of our project. Instead of trying to find a solution together with Dr. Pizka, the mission changed to evaluating the proposed solution. Step one was to see if the suggested solution was suitable for Ericsson's purposes. If the outcome of step one was, positive, then step two would be to implement a solution for the Sparc architecture. While evaluating step one, less than one week after the snapshot was published, Mr. Jelinek published a solution for the Sparc architecture on the GCC patches list. As with Mr. Henderson's patch, some discussion followed this patch. After a week an improved version for the Sparc architecture was published.

As the chapter headline implies it seems like the authors' activity on the mailing lists have really awoken the interest for tail calls and brought forward the long time perceived requirement for a solution for the tail call problem.

3.9 The Solution Provided By Cygnus

As previously mentioned, a solution was implemented in the snapshot egcs-20000313. At this point it seamed to be the best possible outcome of the authors project since persons closely connected to GCC provided the solution. The motivation for this statement is that an Ericsson specific solution produced by the authors might have led to future problems, if the solution would not have been a part of the main trunk, as GCC is continuously developing. The Cygnus solution covers the problem in the test program mentioned in section 3.3. The Cygnus solution was provided by the end of the experiment phase of our project. Since the solution was provided by Cygnus and passed the test program without any problems, we decided that this would be the suitable solution to evaluate in chapter 4.

---

[9] During the time between two official releases, an "unofficial" patched version is available for download from the GCC homepage. This patched version is called a snapshot.

3.10 Chapter Summary

This chapter describes how the authors began the task. In order to obtain adequate knowledge of the problem we studied various sources, i.e. papers, compiler construction books and the Internet. By subscribing to the GCC mailing lists, we have been able to find knowledgeable people. These personal contacts have been a great source of information when it comes to our understanding of the problem and also for the development of a solution to the tail call problem.

We have with the help of code examples exposed the problem with tail calls and also why the use of gotos in the new version of GCC causes problem. It is nowadays considered, by most programmers, that the undisciplined use of gotos may lead to unstructured programs. In versions later than GCC 2.8.1 the use of gotos is considered undefined behavior and **can not** be used for jumps between functions. The optimization is performed “function by function” rather than “statement by statement”. If GCC does not find a corresponding jump to a label within the same function, all code below that label will be classified as dead code and will be removed. If a solution to the tail call problem would be solved, Ericsson could rewrite their simulation program and use tail calls instead of gotos, and hence be able to upgrade their GCC software.

We have found several other related projects. At the School of Computer Science in Toronto, the goal is to introduce CPS for C programmers. A solution to the tail call problem would enable them to reach their goal. Several persons have been involved in mailing list discussions of possible solutions to the tail calls problem. Jakub Jelinek and Jeffrey Law, both had been involved in creating implementations from those suggestions. Dr. Marcus Pizka is involved in the development of the GIC and C-- compiler. The idea of these compilers is to use the GCC backend. The authors and Dr. Pizka cooperated in trying to find a solution to the problem but when the Cygnus patch was published the cooperation was ended.

In order to evaluate the optimization in RTL we used the possibility of obtaining RTL dumps. By comparing dumps from different source codes and different stages of the compilation, we would be able to understand the connection between source, intermediate, and target code.

From the point we joined the mailing lists, the activity and interest for the tail call problem increased. The Cygnus patch was released and our project changed direction. A solution to the tail call problem would not only benefit our project, but also Scheme users.

4 Evaluation

4.1 Introduction

This chapter evaluates the solution provided by Cygnus and Mr. Jelinek. The Cygnus/Jelinek solution to the tail call problem is very detailed. The authors conclude that their experience is insufficient to evaluate the full details of this solution. However, at the same time, we consider that the Cygnus staff, as well as Mr. Jelinek have sufficient experience with GCC and are hence able to provide a reliable solution to the tail call problem. These facts make it difficult to measure the value and correctness of the proposed solution, as far as Ericsson is concerned. Therefore, will this evaluation primarily concentrate on testing the solution on the test program provided.

We will first present each part of the solution and also mention some unexpected side effects. Then we will use the test program provided in order to evaluate the validity of the solution. During the project's evaluation phase, the project's initial premises have been changed. The reason for this change is that the first test program we received from Ericsson did not correspond to the real simulation program. New test programs have been provided during the evaluation phase.

4.2 The Cygnus Solution to the Tail Call Problem

The Cygnus patch is primarily the work of Mr. Law, as mentioned in section 3.4.4, though there are several other Cygnus employees that have contributed to the work. Mr. Henderson's role has been to clean the code up so that it applies to the current GCC tree.

The theory of operation is as follows: for each eligible call, generate three instruction sequences, one for a "normal" call, one for a "sibling" call, and one for tail recursion. These are placed into a CALL_PLACEHOLDER pattern[10] and pass relatively unchanged through inlining. As one of the first optimization passes, the Control Flow Graph[11] for the function is built and simplified. Calls that appear at the end of a block that is a predecessor for exit are replaced by one of the saved tail-call sequences. All other calls get replaced by the saved

[10] A pattern could be described as a rule that describes a set of strings.
[11] A Control Flow Graph is the graph that describes the jumps between blocks of code.

normal call sequence. A new file, *sibcall.c*, was created in order to handle the new data structure CALL_PLACEHOLDER and its different actions. The complete file *sibcall.c* is shown in Appendix B.1. Except from the new file, the Cygnus patch affects the following twelve files: *calls.c, final.c,. flow.c, function.c, genflags.c, integrate.c, jump.c, rtl.c, rtl.h, toplev.c, tree.c* and *tree.h.* Some architecture dependent files are also affected, but since they do not concern our project, we have chosen not to mention them any further. As a whole, the patch consists of more than 4100 rows (~75 pages).

It is not easy, even for a very experienced eye, to interpret a patch because the patch consists only of fragments of the original file, taken out of its context. In order to fully understand a patch, one has to know and understand the original file's code. It would have been interesting if both the Cygnus patch as well as the Jelinek Sparc patch had surfaced at an earlier stage of our project. That would have given us time to really investigate the code and hence be able to evaluate and describe the full details of the patches.

The Cygnus patch does not offer a solution for the future. A long-term, tree-based solution is already planned by the GCC steering committee [URL1]. When the tree-based solution is implemented sometime in the future, it would enable a solution for generic tail call elimination. Multiple return types and variable number of function arguments are some of the additional features that will be supported. We have not found any time schedule or details about the future work with the tree-based solution.

4.3 Mr. Jelinek's Sparc Patch

When Cygnus published their tail call problem-patch, the patch had support for the architectures Alpha, PA, MIPS and x86. Unfortunately Sparc was not supported. Mr. Jelinek reported that he had found some errors in the Cygnus patch and he also suggested some improvements to other flaws that could cause major problems when compiling for Sparc.

After less than a week, Mr. Jelinek had produced and published first version of a Sparc patch[12]. A few days later, after some improvements a final version was published. This patch made it possible for us to test the Cygnus patch on the specified target environment. In order to produce the patch, Mr. Jelinek had to rewrite code in the files *spark.c, spark.h*, *spark.md* and *spark-protos.h.*

The complete patch has a total length of 800 rows and also consists of code correcting some of the remaining errors and flaws in the Cygnus patch. As for the Cygnus patch, it is

difficult to interpret this patch. We will try to in a simple manner describe some of the changes and extensions of the existing file that Mr. Jelinek has performed in the patch. Mr. Jelinek has for instance implemented support for sibling calls by changing the register handling and adding new functions and data types in order to handle sibling calls. When adding new types he had to change some conditions as well. Mr. Jelinek has himself, commented the changes in the patch's code. For those readers with sufficient knowledge in the area, we recommend more in depth study of Appendix C.1.

Since this patch has been implemented in the GCC main trunk, we have no reason to doubt the correctness of this patch. In section 4.5, we will examine if this patch together with the Cygnus patch solves Ericsson's problems with tail calls.

4.4 Unforeseen Problems Caused by the Patches

The patch provided by Cygnus had some negative side effects. Different users around the world have reported these side effects via the GCC mailing lists.

One problem was reported by Jan Hubicka, who experienced a compilation performance slowdown by approximately 15%. Gerald Pfeifer also noted a 10-15% slowdown. Mr. Hubicka has worked hard for a solution to this and other minor problems that might have caused the slowdown. Mr. Hubicka has provided the GCC patches list with several patches to solve the problems. The performance slowdown mainly refers to the new flow graph construction.

Bruno Haible reported debugging problems after installing the new patch. He had problems with getting a full debugging report since the patch is in fact created in order to save space and cut the "connection" between caller and callee. "Many GNU programs are compiled with "-O2 -g" by default, which has been up to now a good compromise between speed and ease of debugging." Mr. Henderson draws the conclusion that Mr. Haible is working on x86 since the problem described is well known for other architectures. According to Mr. Henderson the effect on Sparc is no worse than before. "On alpha and other wide RISC targets the instruction scheduling turned on with -O2 makes debugging quite a challenge most of the time. "

Mr. Jelinek and Bernd Schmidt, each describes the same problem. Mr. Schmidt reports: "The instruction that pushes the address of the label is inside a CALL_PLACEHOLDER, so

---

[12] Mr. Jelinek's patch is shown as an example of a patch in Appendix C.1.

we never notice that the label is used, and turn it into a NOTE_INSN_DELETED_LABEL. This later causes mark_jump_label to call abort." This problem was quickly solved. Mr. Schmidt and Mr. Jelinek each suggested a solution to the problem and Mr. Schmidt's solution was preferred by those responsible at GCC.

4.5 Testing the Patches on Ericsson's Test Program(s)

As a part of verifying the suitability of those patches for Ericsson's purposes, we have used the test programs made available to us by our supervisor at Ericsson. After successfully have tested the Cygnus/Jelinek solution with the test program described in section 4.5.1, we started evaluating the patch. Mr. Einarsson created another test program, this one with function pointers, described in more detail in section 4.5.3.

4.5.1 The Function Call Test Program

In an early stage in this project, a test program was given to us. This test code has been the foundation of the project and has been used as an example in discussions about tail calls both on the Internet and personal contacts. The code has earlier been thoroughly described in section 3.3 and will now be used in order to verify the solution provided by Cygnus/Jelinek.

```
int sibling1(int x1);
int sibling2(int x2);

int main()
{
      int x0 = 1;
      sibling1(x0);
}
int sibling1(int x1)
{
       printf("%i\n",x1);
       x1 = x1+1;
       return sibling2(x1);
}
int sibling2(int x2)
{
      printf("%i\n",x2);
      x2=x2+1;
      return sibling1(x2);
```

Figure 4.1 The function call test program

When we evaluated the provided solution with the function call test program, the outcome was not satisfactory. However, Mr. Einarsson tested the solution on a void function call test program as well and that code worked. The void function call test program will be described in section 4.5.2.

Testing the function call test program, the same segmentation fault error occurred as for the function call program without the new patch. The outcome of the test concerned Mr. Einarsson so he provided us with the void function test program (Figure 4.2). He asked us to investigate the reason why the function call test program did not work, while the void function test program did, and if possible find a solution.

Since Mr. Jelinek was the provider of the Sparc architecture patch, we forwarded Mr. Einarsson' question to him in order to se if it was a Sparc related problem. As usual, Mr. Jelinek answered within a day. He could not see any problem and asked us which grade of optimizing we used, i.e. which flag we used when compiling the function call test program. We were using the '-O3' flag as we had done in all previous test cases. Mr. Jelinek suggested the use of the '-O2' flag instead, which worked out just fine. The 'O3' flag enables optimization by inlining. Inlining is a good solution when a program only has a few function

calls. In the case with tail calls, there are not a few calls, rather the opposite, so ‘O3’ is not possible to use.

After running the test program on the patched GCC snapshot we could draw the conclusion that the patches satisfied all of Ericsson’s requirements for this test program. The tests were performed on the Sparc architecture at Ericsson. We have not taken the other architectures into consideration, although the Cygnus patch does support several others. We have followed the discussion on the mailing lists and there have been very few architecture problems reported due to the new Cygnus tail call patch. Cygnus’ patch combined with Mr. Jelinek’s patch, both improved with minor bug fixes, creates a solution that covers the most commonly used architectures.

At this point Ericsson’s requirements were satisfied. The experiment phase was over and the time limit for the project’s evaluation phase was almost reached. It was time to reach a closure of the evaluation phase and enter the phase of completing the report.

4.5.2 The Void Function Call Test Program

During the evaluation of the function call test program, Mr. Einarsson came up with the idea of testing a void function rather than the non-working function with return values (in this case *int*). The void function call program turned out to work fine The function call test program, slightly modified is shown in Figure 4.2. With this working test program in mind, we contacted Mr. Jelinek.

```
int main()
{
      int x0 = 1;
      sibling1(x0);
}
void sibling1(int x1)
{
       printf("%i\n",x1);
       x1 = x1+1;
       sibling2(x1);
}
void sibling2(int x2)
{
      printf("%i\n",x2);
      x2=x2+1;
      sibling1(x2);
}
```

Figure 4.2 The void function call test program

4.5.3 The Function Pointer Test Program

A couple of weeks later, Mr. Einarsson created yet another test program, containing function pointers, more similar to the simulation program that Ericsson uses in their APZ simulator. At this point we understood that the first test program Mr. Einarsson provided us with, did not correspond to the real program. When we received the first code, the function call test program, Mr. Einarsson told us that the code in the test program was not exactly the same, but a solution for that code would also be a solution for the real simulation program as far as he knew. One can comment here, that the authors' supervisor at Karlstad University, Mr. Ross asked us at an early point, if the test program really was representative for the simulation problem. As far as the authors then knew, this was the case.

When testing the function pointer test program (Figure 4.3) on the snapshot, this new test program caused the same segmentation fault as in GCC without the tail call patch. The conclusion is that the snapshot at this specific point, is not a sufficient solution to Ericsson's problem.

As mentioned earlier, the time limits for our phases "research and background reading" and "evaluation", had expired and there was no time left to further investigate the reason why the snapshot did not satisfy Ericsson's needs.

```
/*FUNCTION DEFINITIONS */
void one(int x);
void two(int x);

void (*onePtr)(int) = &one;
void (*twoPtr)(int) = &two;

int main(void)
{
     onePtr(0);
}
void one(int x)
{
     printf("one %i\n",x);
     twoPtr(x+1);
}
void two(int x)
{
     printf("two %i\n",x);
     onePtr(x+1);
}
```

Figure 4.3 The function pointer test program

4.6 Chapter Summary

A solution to the tail call problem has been presented. The Cygnus patch is the major part of the solution since it is this patch that deals with evaluating whether the function call is a tail call or not. To make this possible *sibcall.c* was created and twelve other files were changed. This patch supports several target architectures, but not Sparc.

The main idea of the solution is to store a function call in the data structure CALL_PLACEHOLDER. Three instruction sequences are generated, one for a "normal" call, one for a "sibling" call, and one for tail recursion. Depending on which kind of call it is, the corresponding instruction sequence is used.

Jacub Jelinek provided a patch with a solution for Sparc. In the content, the propotions of this solution were not as wide ranging as the Cygnus patch, but for our project, the Jelinek patch is very significant.

As always, consequenses of changes in GCC are difficult to predict. The tail call solution caused a variety of minor negative side effects. After a few weeks, a number of negative side effects had been reported, but also, most of them were solved.

As a part of verifying the patches suitability for Ericsson's purposes, the authors have used Ericsson's test program. The verification of the function call test program, which was provided to us in the early stages of the project, showed that the program fulfilled all requirements. The void function call test program was also verified to be correct.

The function pointer test program has not yet passed the verification. However, the Cygnus/Jelinek solution has brought Ericsson much closer to a final solution to their problem.

5 Conclusion

5.1 Introduction

This chapter will describe the achievements arrived at in this project. The issues of background reading and how the timetable was used in order to optimize each phase of the project will also be shown.

The interesting developments in the GCC compiler and how a single user can make a difference to such changes will be presented as well as the experiences the authors have gained during this project. The most important issue of a project will be described in the final sections *5.7 What Was Achieved*, 5.8 *What Could Have Been Done Differently?* and 5.9 *Recommendations for Further Investigations on the Subject*.

5.2 Background Reading for the Project

This section will concentrate on the issue of how the background reading was performed. Could we have performed the background reading in another way? Due to our prior experience in compiler construction, we needed to do a great deal of reading about compilers in general and GCC in particular in order to be able to understand the width of the tail call problem. Considering the complexity of GCC, the relatively short time the schedule allowed for background reading was not quite enough to fully understand all details of GCC. Nevertheless, the background reading was very valuable and helped us to better comprehend details of GCC as the project proceeded.

5.3 The Timetable and the Project Phases

The figure below will give the reader an overview of our timetable for this project. As one can see in the timetable, apart from the documentation there are five other phases in the project. Our objective was to perform documentation continuously as the project proceeded.

Month	January			February				March				April					May				June
Week	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Activity

Research and background reading

Evaluation

Construct, implement and testing

Documentation

Finalizing a prel. version

Extra time

Finalizing the report

Prep. for opposition

Figure 5.1 Timetable

The other phases of the project are:

- Research and background reading. During this eight-week phase we read background material and searched the Internet for useful information about the subject and related topics. When we reached week ten, the Cygnus patch was released. The following week Mr. Jelinek released his Sparc patch. These two events prolonged the research and background reading phase.
- Evaluation. After the release of the two patches, the evaluation phase changed from evaluating suggested solutions to evaluating an already implemented one. Due to the implemented solution and the new Ericsson test programs, this phase was also extended by a few weeks.
- Construction, implementation and testing. Ericssons's requirements resulted in this phase becoming optional. If we in an early stage of the project could see a possibility of implementing a suitable solution, this phase would be carried out. The phase began as we established contact with Dr. Pizka. As mentioned earlier he had agreed to guide us through an implementation. Implementation and documentation of such a solution would have taken more time than the timetable allowed. Hence an agreement was reached with Ericsson, to design the solution during the remainder of the construction, implementation and testing phase. The predominant part of the implementation and testing could be

carried out as a separate summer project. This phase was discontinued when the Cygnus solution became public.

- Completing the report. Planning and layout of the report had to be altered when the unexpected release of the Cygnus/Jelinek solution appeared. Instead of designing a solution together with Dr. Pizka, we decided that the most appropriate solution was to resume the work of evaluating and documenting solutions. This time we had an already implemented solution to work with.

5.4 Writing the Report

A few weeks into the "research and background reading" phase we began to write our project report. Prior to the project, we had to write a short summary about the project and hand this in to Karlstad University. We used this summary as a foundation for the background chapter in our report. When writing the report we have used a Swedish-English lexicon [PET89]. We have also used both an online Swedish-English lexicon [9] and an online English dictionary [10], which has both proved very useful and timesaving.

5.5 The Dynamic Evolution of GCC

The development and maintenance of GCC is a dynamic process that proceeds continuously with users all over the world affecting GCC's evolution. When one comes to work in the morning, there is always a chance that a new patch has been posted to the mailing lists. Depending on which participants are active on the lists, their questions and their answers, the GCC development can follow a certain direction. Figure 5.2 will illustrate the dynamic influences of the GCC evolution as we see it.

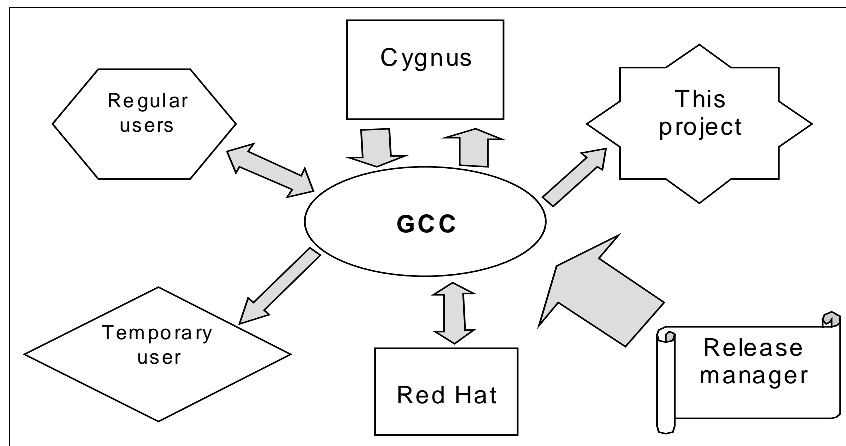


Figure 5.2 Actors influencing GCC evolution

The GCC release manager is responsible for the direction of the GCC development and answers to the GCC steering committee and its guidelines. The release manager has the authority to support or reject suggestions or patches that is posted on the mailing lists. GCC could be seen as a tree with a trunk and branches. Figure 5.3 will illustrate how the GCC tree could look like at a specific moment.

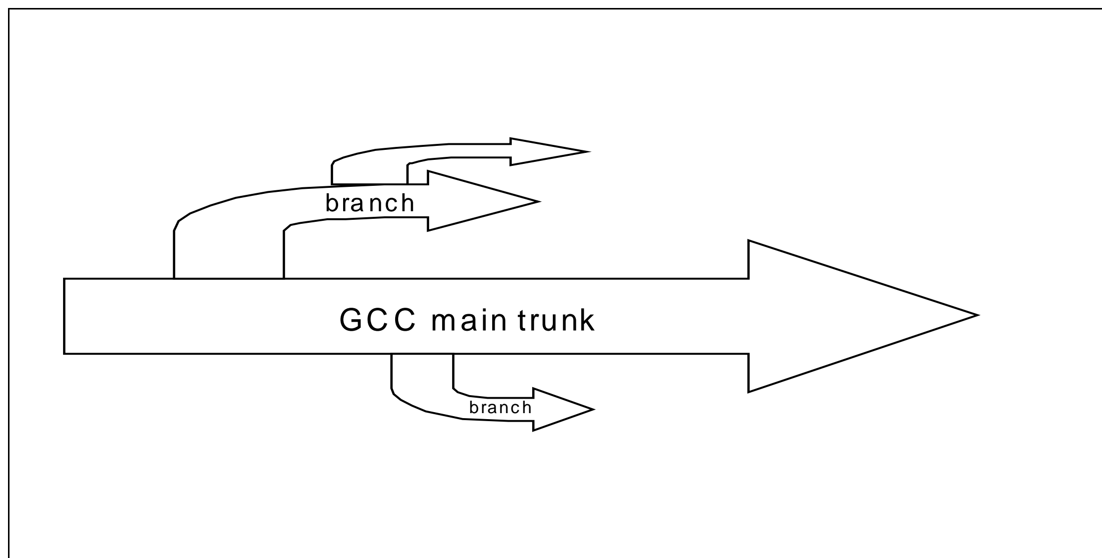


Figure 5.3 The GCC tree

If the patch follows the guidelines for the trunk, it can be accepted and become a part of the official version of GCC. Even if a patch does not follow the guidelines for the trunk, it could be very useful for many users. The approach when this occurs is to allow a branch to grow out from the trunk. There could be several branches alive at a particular time. Later updates in the trunk could support the requirements that caused a branch. When there is no longer a need for a branch one can prune the branch and return to the trunk again.

When a subject is heavily discussed on the mailing lists and the need for a solution is desirable, GCC staff takes an active interest in finding an appropriate solution. As an example of this, it seems like the authors' activity on the mailing lists aroused the interest of tail call elimination within the GCC organization. Less than two months after we first addressed the mailing lists, a solution supported by GCC was published.

5.6 Experiences Gained

During the project, the authors have gained increased knowledge in several areas such as compiler development, the Unix operating system, project management and writing technical English. Further more we have gained new personal contacts and also knowledge as how to extend our personal network via the Internet. We will discuss several of the areas below:

- *GCC*. The area that has obviously engaged us the most is the compiler system GCC. We had not understood earlier, the size and complexity of the GCC compiler system. After the background research, we realized that we had a task of considerable proportion in front of us. We have also learned how the development and maintenance of GCC is a dynamic process, involving a large amount of interested parties. This dynamic process has been discussed in section 5.5. We have learned that GCC is a compiler with a frontend that support multiple high level programming languages, and that GCC also has a backend for generating and optimizing machine dependent code for various hardware architectures. Closely interwoven with the backend, there is a Lisp like intermediate language, the Register Transfer Language (RTL). It is in the RTL stage that nearly all optimization is performed. We have understood that GCC, during compilation, involves a number of different files in the different passes that is needed in order to complete the compilation. These passes have been introduced to the reader in section 2.6.1.

- *Unix*. The author's experience of the Unix operating system prior to this project was limited. It would have been valuable to the initial phase of the project if we had taken the course "C and Unix" at the University last semester. Our experience had come from those courses at Karlstad University, where the Linux operating system was used. The experiences we have gained throughout this project are in file management, different printing commands and other essential Unix commands.
- *Compilers*. During our background research we learned that there are a large number of compilers for different uses. Since our main concern has been GCC, we have chosen not to look into details about any other specific compilers than GCC. The overview we acquired from the various compiler systems though, helped us to understand how a compiler in general works. By looking at other compilers, one can obtain a different view of sections in the GCC documentation that are difficult to understand, and hence be able to better understand a GCC phase or a section in the GCC manual. Looking at those different compilers[13] has helped us to understand the passes that are normally performed in a general compiler.
- *Project management*. Earlier, we have only worked together in short term projects. A project, such as this is a new experience when it comes to having a client to continuously report to, and a time schedule of more than four months to take into consideration. When it comes to this project's time schedule we think that we have succeeded in following the time schedule and its time limits. In a project, it is important to keep a record of what has been done and take notes about events as they happen. It is also important to have a version number system for source code and other documents.
- *Technical English*. The multinational company Ericsson has the policy of using English as business language. Due to this policy they wanted us to write the project report in English. All information gathering via the Internet, both reading and personal contacts have required a good knowledge of English. Also, the vast majority of literature in the area of compilers and compiler construction is in English, so it is not an understatement to say that we have been deeply involved in learning more about reading and writing technical English. In doing so, our advisor Donald F. Ross has been a great resource when it comes to giving the report its final touch. There are not

[13] The studied compilers are XMPL [LUN91], "An example compiler" [URL5], "A simple compiler" [AHOS86] and gic [PIZ97]

many groups that have had the opportunity to have such an experienced adviser when it comes to English grammar. As a conclusion, all the reading and writing technical English throughout the project has developed our ability to read, analyze and rephrase English technical literature.

- *Ericsson*. The opportunity to work for Ericsson a whole semester has given us insight into working for a multinational company. Despite Ericsson’s size, one has the illusion of working in a small scale, intimate environment. At TSP, they took time in order to inform us about the structure of the department and the phases of one of their projects. We had also the opportunity to follow and learn the daily routines of TSP.
- *People available on the web*. When searching for information, we have learnt that there are many people willing to help, once you have found the right place to look for these people. Mailing lists of various kinds are often an excellent source of information. People, active on a mailing list are often very skilled and knowledgeable in their particular area and see the possibility of personal development by helping others. All of the author’s questions have met with a positive reception on the GCC mailing lists. Therefore we can recommend this way of searching and gaining knowledge.

5.7 What Was Achieved

This section describes the most important benefits of the project *Tail Call Elimination in GCC*. The project’s primary goal was to satisfy the needs of Ericsson. Since GCC is a world wide commonly used compiler, all improvements as a result of this project would also be of gain for a large number of other users.

According to Ericsson, our project has accelerated the development of a solution for the tail call problem. When Mr. Einarsson posted his tail call questions on the GCC mailing list, he met with a positive response. There were some suggestions as how one can approach the problem but nobody seemed interested in beginning the task themselves.

These statements strengthen and supports the authors’ own opinion in the matter: The project *Tail Call Elimination in GCC* has indeed accelerated the development of a solution for the tail call problem.

5.7.1 Non-Ericsson Specific Benefits

Some of the benefits from this project will be useful for other GCC users and developers besides Ericsson.

- Support for tail call elimination on several architectures; MIPS, PA, Alpha and x86. The Cygnus patch is integrated in the GCC tree and people closely connected to GCC have created the patch. These facts emphasize the strength of the solution.
- Scheme users have benefited from the project. Since some of the Scheme-to-C compilers typically call continuations with no parameters, tail call elimination would turn all of those calls to jumps for a significant improvement in performance [MAS99].
- The related project presented in section 3.4.1 has the possibility to resume their work in introducing CPS for C-programmers. The main problem in their project was the lack of support for tail call elimination in C compilers and especially GCC. This obstacle has now been eliminated.
- Facilitate the implementation of the C-- compiler. For those (including Dr. Pizka) involved in the work of creating a language better suited as an intermediate language than C [PIZ00].
- Cygnus' "selected customers" that Mr. Law had supplied with a GCC-tree-branch solution have a possibility to swap to the trunk solution. With the trunk solution they can obtain support from a wider sphere of developers and also update their software continuously if they want.

5.7.2 Benefits to Ericsson

The assignment was to investigate if it is possible to implement tail call elimination in GCC 2.95 and if so, implement a solution for the Sparc architecture. If possible a hardware generic solution was to prefer.

It did not take long before we could give an answer to the first part of the assignment. After some background reading it became obvious that it, with no doubt, was possible to implement tail call elimination in GCC.

The second part of the project was to implement a solution for the Sparc architecture. When Cygnus and Mr. Jelinek released their patches, such an implementation existed. The Cygnus/Jelinek solution also fulfilled the additional requirement of the second part of the assignment since the most commonly used architectures are supported.

As mentioned in section 4.5, the solution satisfies Ericsson's requirements for the first two test programs. Unfortunately, the solution is not sufficient for the function pointer test program. However, this project has brought Ericsson much closer to a solution that would handle the remaining problem. Several sub-goals have been achieved.

- As far as our supervisor and we have understood, Mr. Jelinek's Sparc patch, as it looks today, needs little, or no further development in order to solve the remaining problem.
- The scope of the Cygnus patch leads to the fact that the extent of any remaining work is quite limited compared to the original situation.
- The authors' personal contacts during this project can be used if Ericsson wants to proceed in any further investigations.
- The source material, which the authors have assembled at Ericsson, is a resource for further investigations.

5.8 What Could Have Been Done Differently?

In summarizing a project, one can always say that there are certain aspects that might have been done differently. This project is of course no exception. We have, during evaluation found some issues in our project that, if done differently, could have altered the outcome of the project.

- The connection between the function call test program and the real simulation program should have been verified in an early stage of the project. Had this been carried out, the outcome of the project might have been different.
- The GCC manual is an excellent source of information. If we, in an early stage of the project had known just how much information we could obtain from the manual, we would have paid the manual more attention at this stage. We did though, study the manual at an early stage, but more as an overview. A deeper exploration was performed during our cooperation with Dr. Pizka. The conclusion is that we could have performed a more in depth study of the GCC manual at the early stages of the project, but the question is: Would we, as inexperienced in the compiler area, have benefited from the detailed information in the early stages of the project?

- Could we have been more active on the mailing lists? Of course, but the conclusion is similar to the one above. Did we have the knowledge to ask adequate questions, and did we have the ability to interpret the answers? Probably not in the earlier stages of the project.
- We could have worked more with writing the report earlier. By doing so, an earlier evaluation of the accumulated material would have been performed. The understanding of the material would also benefit from the early writing. The drawback of allocating time for writing is that we would have lost the same amount of time for further investigations and background reading.
- Looking back, an office of our own would have facilitated the work. For a couple of months we had the office printer in our room. Sometimes it could be disturbing when people collected their printouts. During the entire project, we shared office with another group from Karlstad University.

5.9 Recommendations for Further Investigations in This Area

Some of the material in this section springs from information, which we have received while finalizing the report. Since our recommendations to Ericsson partly depend on this last minute information, we have chosen to include the material even though the information is not previously mentioned in the report.

As we were finalizing this report, we have also tried to investigate how much work has to be done in order to adjust the existing GCC version, (with the Cygnus/Jelinek patches committed) to solve Ericsson's tail call problem. We have conferred with our knowledgeable contacts [PC3, PC4, PC5]. "…How much work remains before one can get the function pointer test program to work?…" Their summarized opinions in the matter are as follows:

Jan Hubicka:

Basically you are asking here for indirect tail calls. The main problem is that you need register for destination address. This register must be caller saved, since it must be live after epilogue. Second problem (that appears at Sparc) are register windows. Basically it is not too hard to implement this - you need to add new a constraint in *sparc.md* for caller saved registers, add pattern and enable this stuff in *calls.c*. I think that with expectable problems related to everything in *calls.c* area it may take about two days to add. On the other hand, I am

not sure that this optimization will ever save so much CPU time. Calling indirectly is not too common and calling indirectly in tail call sequence even less.

Jeffrey Law:

Optimizing tail call sequences for indirect calls is not likely to work anytime soon. There are some particularly tricky issues that would have to be resolved before it could work.

Richard Henderson:

That depends. It's not solvable in general, only for specific targets. For instance:

(1) Alpha can only perform tail call elimination to functions that use the same GP. When calling through a pointer, we cannot prove that will be true. This is fallout from the calling conventions -- on return from a function, the GP must be correct for that function.

(2) ARM has 4 call-clobbered registers, all of which are used for passing arguments. So if the target function needs more than three arguments, it's not possible to tail-call through a pointer.

(3) i386, with -fpic, requires %ebx to contain a pointer to the function's GOT during function calls. Since %ebx is a call-saved register, we cannot make tail calls of any kind.

For other targets, it suffices to define a register class, which contains all of the call-clobbered non-argument registers, and have the sibcall pattern accept only that register class. It would take about a day to implement for any one target.

After our contacts with these people, it has become clear that they refer to the call convention used by Ericsson as, *tail calls through pointers*, or *indirect tail calls*. If we trust Mr. Henderson and Mr. Hubicka, a solution that satisfies Ericsson's requirements would only take an experienced compiler developer a couple of days to implement.

Mr. Law's statement might be interpreted as: "At the moment, Cygnus has other projects with higher priority." If Cygnus where to participate in developing a solution, the solution would be integrated in the GCC main trunk and contributed to the GCC community. Ericsson would obtain support from a wider sphere of developers and also be able to update their software continuously if they want.

Since we did not achieve our goal one hundred percent, this report can serve as a springboard for further work. These are our recommendations as to what Ericsson can do in order to obtain an implementation that solves their tail call problem:

- *Wait and see*. A future tree-based solution is planned. This could provide Ericsson with a final solution to their problem.

  Advantages: No development cost. No risk for developing a product that could be out-of-date in a near future. No loss of performance. Someone else could solve the problem.

  Disadvantages: Not upgrading the software is not a good solution. The development of software and hardware is a constantly ongoing parallel process. Old software and new hardware does not always go together.
- *In-house solution*. Use this report, and the accumulated material at Ericsson, as a springboard for further work. A good idea is also to contact the people that have been involved in the development of the Cygnus/Jelinek solution, and to whom the authors have established contact. E-mail addresses could be found under the section Personal Contacts in the chapter References.

  Advantages: If there are persons at Ericsson, with the sufficient knowledge, such a solution would be less expensive and also custom made for Ericsson's needs.

  Disadvantages: An in-house solution would probably not belong to the GCC trunk and hence difficult to support. A similar situation as we have today can occur. A future change in the GCC trunk can cause new unforeseen problems, as with the gotos.
- *Purchase a solution*. There are several companies that supply GCC support and hence are possible contractors for the remaining work.

  Codesorcery. It is possible to contact them via info@codesource.com.

  Cygnus. According to Mr. Law, sales@cygnus.com would be a proper starting point. However, Cygnus does not accept anything less than four weeks of work.

  SuSE. Mr. Hubicka, working for them, says that SuSE definitely can provide a solution. They can be contacted via info@suse.de.

  RedHat. Mr. Jelinek's employer. We did not find any suitable e-mail address. Instead we refer to www.redhat.com.

Advantages: Such a solution would probably be a part of the GCC trunk and hence easier to support and upgrade. The time it would take to implement the solution is probably shorter.

Disadvantages: Probably a more expensive solution.

Before Ericsson decides which recommendation they want to pursue, there are some facts to take into consideration:

- Examine the possibility to avoid the dead code elimination in GCC 2.95. This could lead to a possibility to use gotos in this version.
- Is it worth waiting for a tree-based solution, or even financially support the development of a tree-based solution? If so, this might be a long-term solution for Ericsson’s needs.

5.10 Final Words

This has been an interesting project. We have brought Ericsson closer to a solution to the tail call problem. The authors have gained new experience in several different areas and we are very satisfied with the outcome of this project.

Our achievements could not have been possible without the help and understanding from the following persons:

Donald. F Ross, our advisor at Karlstad University.
Magnus Einarsson, our supervisor at Ericsson Infotech AB.
Patrick Carlén, Ericsson Infotech AB.
Dr. Markus Pizka, our guide in the GCC jungle.
Peter and Johan, roommates at Ericsson.
All the employees at TSP, Ericsson Infotech AB.
All the helpful persons we met on the Internet.

Last but not least, our families who have endured the privation we have caused them.
Lena and Simon
Kerstin and Lars
Ewa and Tomas
and little Trolle

Thank you all!

References

Personal Contacts

[PC1] Dr Markus Pizka. marcus.pizka@softwareag.com

[PC2] Jacub Jelinek. jakub@redhat.com

[PC3] Jeffrey Law. law@cygnus.com

[PC4] Richard Henderson. rth@cygnus.com

[PC5] Jan Hubicka. jh@suse.cz

[PC6] Jörgen Sigvardsson. jorgen.sigvardsson@kau.se

[PC7] Magnus Einarsson. Magnus.einarsson@ks.ein.se

Books and Papers

[AHOS86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman Compilers - *Principles, Techniques, and Tools*. Addison Wesley 1986.

[HOG00] Johan Högberg, *SimZOY technology,* Ericsson Infotech AB, 2000.

[LUN91] Hans Lunell. *Kompilatorkonstruktion i teori och praktik*. Studentlitteratur 1991.

[MAS99] David V. Mason. *Continuation Passing Style in C and the Cost of Tail-call Elimination With GCC.* School of Computer Science, Ryerson Polytechnic University. March 1999.

[MUC97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 1997.

[PIZ97] Markus Pizka. *Design and Implementation of the GNU INSEL-Compiler gic.* Technische Universität München, Institut für informatik, 1997.

[PIZ00] Markus Pizka. *The Portable Assembly Language C--: A Critacal Review and a GCC Based Prototype.* Microsoft Research, Cambridge. February 2000.

[STA99] Richard M. Stallman. *Using and Porting the GNU Compiler Collection.* Free Software Foundation. July 1999.

[WES92] Anthony Weston. *A Rulebook for Arguments*. Hackett Publishing Company, 2nd edition, 1992.

[PET89] Petti et. als. *Stora svensk.engelska ordboken.* Esselte studium AB. 1989.

URLs

[URL1] http://www.fsf.org/software/gcc/gcc.html **(00-03-06. 15:32)**

[URL2] http://gcc.gnu.org/ml/gcc/ **(00-03-06. 15:32)**

[URL3] http://gcc.gnu.org/ml/gcc-patches/ **(00-03-06. 15:32)**

[URL4] http://wwwspies.informatik.tu-muenchen.de/personen/pizka **(00-03-06. 15:32)**

[URL5] http://www.iecc.com/compilers/crenshaw **(00-03-06. 15:32)**

[URL6] http://www.nightflight.com/foldoc/index.htm **(00-03-06. 15:32)**

[URL7] http://www.suif.stanford.edu/~diwan/243/ **(00-03-06. 15:32)**

[URL8] http://www.cs.nwu,edu/groups/su/resources.html **(00-03-06. 15:32)**

[URL9] http://www-lexikon.nada.kth.se/skolverket/sve-eng.html **(00-03-06. 15:32)**

[URL10] http://websters.searchopolis.com/ **(00-03-06. 15:32)**

[URL11] http://www.ericsson.se/pressroom/comp_newtw.shtml **(00-03-06. 15:32)**

[URL12] http://www.ericsson.se/infotech/ company **(00-03-06. 15:32)**

[URL13] http://bokhyllan.ks.ericsson.se/ **(00-03-06. 15:32)**

[URL12] is not available outside Ericsson.

A Appendix. Influence of Optimization Grades on RTL Code

When the project changed direction from designing a solution to evaluating the Cygnus/Jelinek solution, we decided to cease the evaluation of RTL code. Despite this, we consider that it would be interesting for the reader to see the RTL code. The following pages will give an example of how a user can choose different levels of optimization and how this affects the RTL code for the source code below. Chapter 15 in the GCC manual [STA99] describes the RTL instructions in an informative way.

```
int main()
{
      int x0 = 0;
      sibling1(x0);
}
int sibling1(int x1)
{
       x1 = x1+1;
       printf("sib 2 %d",x1);
       return sibling2(x1);
}
int sibling2(int x2)
{
      x2=x2+1;
      printf("sib 2 %d",x2);
      return sibling1(x2);
}
```

A.1 Without Any Optimization

The example's RTL code when the '-O0' flag is used.

```
;; Function main

(note 2 0 3 "" NOTE_INSN_DELETED)

(note 3 2 6 "" NOTE_INSN_FUNCTION_BEG)

(note 6 3 9 ef541ae0 NOTE_INSN_BLOCK_BEG)

(insn 9 6 12 (set (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int -20 [0xffffffec])) 0)
        (const_int 0 [0x0])) -1 (nil)
    (nil))

(insn 12 9 13 (set (reg:SI 8 %o0)
        (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int -20 [0xffffffec])) 0)) 112 {*movsi_insn} (nil)
    (nil))

(call_insn 13 12 15 (parallel[
            (set (reg:SI 8 %o0)
                (call (mem:SI (symbol_ref:SI ("sibling1")) 0)
```

```
                    (const_int 0 [0x0])))
            (clobber (reg:SI 15 %o7))
        ] ) -1 (nil)
    (nil)
    (expr_list (use (reg:SI 8 %o0))
        (nil)))

(note 15 13 16 ef541ae0 NOTE_INSN_BLOCK_END)

(note 16 15 18 "" NOTE_INSN_FUNCTION_END)

(insn 18 16 20 (clobber (reg/i:SI 24 %i0)) -1 (nil)
    (nil))

(insn 20 18 0 (use (reg/i:SI 24 %i0)) -1 (nil)
    (nil))

;; Function sibling1

(note 2 0 4 "" NOTE_INSN_DELETED)

(insn 4 2 5 (set (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int 68 [0x44])) 0)
        (reg:SI 24 %i0)) 112 {*movsi_insn} (nil)
    (nil))

(note 5 4 8 "" NOTE_INSN_FUNCTION_BEG)

(note 8 5 11 ef520720 NOTE_INSN_BLOCK_BEG)

(insn 11 8 13 (set (reg:SI 106)
        (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int 68 [0x44])) 0)) 112 {*movsi_insn} (nil)
    (nil))

(insn 13 11 15 (set (reg:SI 107)
        (plus:SI (reg:SI 106)
            (const_int 1 [0x1]))) -1 (nil)
    (nil))

(insn 15 13 17 (set (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int 68 [0x44])) 0)
        (reg:SI 107)) 112 {*movsi_insn} (nil)
    (nil))

(insn 17 15 18 (set (reg:SI 108)
        (high:SI (symbol_ref:SI ("*.LLC0")))) -1 (nil)
    (nil))

(insn 18 17 20 (set (reg:SI 8 %o0)
        (lo_sum:SI (reg:SI 108)
            (symbol_ref:SI ("*.LLC0")))) -1 (nil)
    (nil))

(insn 20 18 21 (set (reg:SI 9 %o1)
        (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int 68 [0x44])) 0)) 112 {*movsi_insn} (nil)
    (nil))

(call_insn 21 20 24 (parallel[
            (set (reg:SI 8 %o0)
                (call (mem:SI (symbol_ref:SI ("printf")) 0)
                    (const_int 0 [0x0])))
            (clobber (reg:SI 15 %o7))
        ] ) -1 (nil)
    (nil)
    (expr_list (use (reg:SI 9 %o1))
        (expr_list (use (reg:SI 8 %o0))
            (nil))))

(insn 24 21 25 (set (reg:SI 8 %o0)
        (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int 68 [0x44])) 0)) 112 {*movsi_insn} (nil)
    (nil))

(call_insn 25 24 27 (parallel[
            (set (reg:SI 8 %o0)
```

```
                (call (mem:SI (symbol_ref:SI ("sibling2")) 0)
                    (const_int 0 [0x0])))
            (clobber (reg:SI 15 %o7))
        ] ) -1 (nil)
    (nil)
    (expr_list (use (reg:SI 8 %o0))
        (nil)))

(insn 27 25 29 (set (reg:SI 109)
        (reg:SI 8 %o0)) -1 (nil)
    (nil))

(insn 29 27 30 (set (reg/i:SI 24 %i0)
        (reg:SI 109)) -1 (nil)
    (nil))

(jump_insn 30 29 31 (set (pc)
        (label_ref 37)) -1 (nil)
    (nil))

(barrier 31 30 33)

(note 33 31 37 ef520720 NOTE_INSN_BLOCK_END)

(code_label 37 33 38 3 "" "" [num uses: 1])

(insn 38 37 0 (use (reg/i:SI 24 %i0)) -1 (nil)
    (nil))

;; Function sibling2

(note 2 0 4 "" NOTE_INSN_DELETED)

(insn 4 2 5 (set (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int 68 [0x44])) 0)
        (reg:SI 24 %i0)) 112 {*movsi_insn} (nil)
    (nil))

(note 5 4 11 "" NOTE_INSN_FUNCTION_BEG)

(insn 11 5 13 (set (reg:SI 106)
        (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int 68 [0x44])) 0)) 112 {*movsi_insn} (nil)
    (nil))

(insn 13 11 15 (set (reg:SI 107)
        (plus:SI (reg:SI 106)
            (const_int 1 [0x1]))) -1 (nil)
    (nil))

(insn 15 13 17 (set (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int 68 [0x44])) 0)
        (reg:SI 107)) 112 {*movsi_insn} (nil)
    (nil))

(insn 17 15 18 (set (reg:SI 108)
        (high:SI (symbol_ref:SI ("*.LLC0")))) -1 (nil)
    (nil))

(insn 18 17 20 (set (reg:SI 8 %o0)
        (lo_sum:SI (reg:SI 108)
            (symbol_ref:SI ("*.LLC0")))) -1 (nil)
    (nil))

(insn 20 18 21 (set (reg:SI 9 %o1)
        (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int 68 [0x44])) 0)) 112 {*movsi_insn} (nil)
    (nil))

(call_insn 21 20 24 (parallel[
            (set (reg:SI 8 %o0)
                (call (mem:SI (symbol_ref:SI ("printf")) 0)
                    (const_int 0 [0x0])))
            (clobber (reg:SI 15 %o7))
        ] ) -1 (nil)
    (nil)
    (expr_list (use (reg:SI 9 %o1))
```

```
        (expr_list (use (reg:SI 8 %o0))
            (nil))))

(insn 24 21 25 (set (reg:SI 8 %o0)
        (mem/f:SI (plus:SI (reg:SI 30 %fp)
                (const_int 68 [0x44])) 0)) 112 {*movsi_insn} (nil)
    (nil))

(call_insn 25 24 27 (parallel[
            (set (reg:SI 8 %o0)
                (call (mem:SI (symbol_ref:SI ("sibling1")) 0)
                    (const_int 0 [0x0])))
            (clobber (reg:SI 15 %o7))
        ] ) -1 (nil)
    (expr_list:REG_EH_REGION (const_int 0 [0x0])
        (nil))
    (expr_list (use (reg:SI 8 %o0))
        (nil)))

(insn 27 25 29 (set (reg:SI 109)
        (reg:SI 8 %o0)) -1 (nil)
    (nil))

(insn 29 27 30 (set (reg/i:SI 24 %i0)
        (reg:SI 109)) -1 (nil)
    (nil))

(jump_insn 30 29 31 (set (pc)
        (label_ref 36)) -1 (nil)
    (nil))

(barrier 31 30 36)

(code_label 36 31 37 4 "" "" [num uses: 1])

(insn 37 36 0 (use (reg/i:SI 24 %i0)) -1 (nil)
    (nil))
```

A.2 With Optimization

The example code’s RTL code when the ‘-O2’ flag is used.

```
;; Function main

(note 2 0 3 "" NOTE_INSN_DELETED)

(note 3 2 6 "" NOTE_INSN_FUNCTION_BEG)

(note 6 3 30 ef520b20 NOTE_INSN_BLOCK_BEG)

(note 30 6 9 [bb 0] NOTE_INSN_BASIC_BLOCK)

(insn 9 30 16 (set (reg/v:SI 106)
        (const_int 0 [0x0])) -1 (nil)
    (nil))

(insn 16 9 17 (set (reg:SI 24 %i0)
        (reg/v:SI 106)) -1 (nil)
    (nil))

(call_insn/j 17 16 18 (parallel[
            (set (reg:SI 24 %i0)
                (call (mem:SI (symbol_ref:SI ("sibling1")) 0)
                    (const_int 0 [0x0])))
            (return)
        ] ) -1 (nil)
    (nil)
    (expr_list (use (reg:SI 24 %i0))
        (nil)))

(barrier 18 17 22)
```

```
(note 22 18 24 "" NOTE_INSN_DELETED)

(note 24 22 0 ef520b20 NOTE_INSN_BLOCK_END)

;; Function sibling1

(note 2 0 61 "" NOTE_INSN_DELETED)

(note 61 2 4 [bb 0] NOTE_INSN_BASIC_BLOCK)

(insn 4 61 5 (set (reg/v:SI 106)
        (reg:SI 24 %i0)) -1 (nil)
    (nil))

(note 5 4 8 "" NOTE_INSN_FUNCTION_BEG)

(note 8 5 11 ef521b40 NOTE_INSN_BLOCK_BEG)

(insn 11 8 25 (set (reg/v:SI 106)
        (plus:SI (reg/v:SI 106)
            (const_int 1 [0x1]))) -1 (nil)
    (nil))

(insn 25 11 26 (set (reg:SI 111)
        (high:SI (symbol_ref:SI ("*.LLC0")))) -1 (nil)
    (nil))

(insn 26 25 28 (set (reg:SI 110)
        (lo_sum:SI (reg:SI 111)
            (symbol_ref:SI ("*.LLC0")))) -1 (nil)
    (nil))

(insn 28 26 30 (set (reg:SI 8 %o0)
        (reg:SI 110)) -1 (nil)
    (nil))

(insn 30 28 31 (set (reg:SI 9 %o1)
        (reg/v:SI 106)) -1 (nil)
    (nil))

(call_insn 31 30 32 (parallel[
            (set (reg:SI 8 %o0)
                (call (mem:SI (symbol_ref:SI ("printf")) 0)
                    (const_int 0 [0x0])))
            (clobber (reg:SI 15 %o7))
        ] ) -1 (nil)
    (nil)
    (expr_list (use (reg:SI 9 %o1))
        (expr_list (use (reg:SI 8 %o0))
            (nil))))

(note 32 31 39 "" NOTE_INSN_DELETED)

(insn 39 32 40 (set (reg:SI 24 %i0)
        (reg/v:SI 106)) -1 (nil)
    (nil))

(call_insn/j 40 39 41 (parallel[
            (set (reg:SI 24 %i0)
                (call (mem:SI (symbol_ref:SI ("sibling2")) 0)
                    (const_int 0 [0x0])))
            (return)
        ] ) -1 (nil)
    (nil)
    (expr_list (use (reg:SI 24 %i0))
        (nil)))

(barrier 41 40 49)

(note 49 41 55 "" NOTE_INSN_DELETED)

(note 55 49 0 ef521b40 NOTE_INSN_BLOCK_END)

;; Function sibling2

(note 2 0 60 "" NOTE_INSN_DELETED)
```

```
(note 60 2 4 [bb 0] NOTE_INSN_BASIC_BLOCK)

(insn 4 60 5 (set (reg/v:SI 106)
        (reg:SI 24 %i0)) -1 (nil)
    (nil))

(note 5 4 11 "" NOTE_INSN_FUNCTION_BEG)

(insn 11 5 25 (set (reg/v:SI 106)
        (plus:SI (reg/v:SI 106)
            (const_int 1 [0x1]))) -1 (nil)
    (nil))

(insn 25 11 26 (set (reg:SI 111)
        (high:SI (symbol_ref:SI ("*.LLC0")))) -1 (nil)
    (nil))

(insn 26 25 28 (set (reg:SI 110)
        (lo_sum:SI (reg:SI 111)
            (symbol_ref:SI ("*.LLC0")))) -1 (nil)
    (nil))

(insn 28 26 30 (set (reg:SI 8 %o0)
        (reg:SI 110)) -1 (nil)
    (nil))

(insn 30 28 31 (set (reg:SI 9 %o1)
        (reg/v:SI 106)) -1 (nil)
    (nil))

(call_insn 31 30 32 (parallel[
            (set (reg:SI 8 %o0)
                (call (mem:SI (symbol_ref:SI ("printf")) 0)
                    (const_int 0 [0x0])))
            (clobber (reg:SI 15 %o7))
        ] ) -1 (nil)
    (nil)
    (expr_list (use (reg:SI 9 %o1))
        (expr_list (use (reg:SI 8 %o0))
            (nil))))

(note 32 31 39 "" NOTE_INSN_DELETED)

(insn 39 32 40 (set (reg:SI 24 %i0)
        (reg/v:SI 106)) -1 (nil)
    (nil))

(call_insn/j 40 39 41 (parallel[
            (set (reg:SI 24 %i0)
                (call (mem:SI (symbol_ref:SI ("sibling1")) 0)
                    (const_int 0 [0x0])))
            (return)
        ] ) -1 (nil)
    (expr_list:REG_EH_REGION (const_int 0 [0x0])
        (nil))
    (expr_list (use (reg:SI 24 %i0))
        (nil)))

(barrier 41 40 49)

(note 49 41 0 "" NOTE_INSN_DELETED)
```

B Appendix. The Cygnus Patch

This patch is quite extensive. It contains more than 4000 rows (> 70 pages), which is approximately the same amount as the entire report. We have chose not to attach the whole patch. If the interested reader would like to see a whole patch, we recommend section C.1. The file *sibcall.c* is a part of the Cygnus patch and will be presented in section B.1.

B.1 The Added File sibcall.c

On the following pages we will present the added file sibcall.c. Except from the changes in twelve existing files, this file had to be added to GCC in order to obtain a better structure when solving the tail call problem.

```
/* Generic sibling call optimization support
   Copyright (C) 1999, 2000 Free Software Foundation, Inc.

This file is part of GNU CC.

GNU CC is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

GNU CC is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with GNU CC; see the file COPYING.  If not, write to
the Free Software Foundation, 59 Temple Place - Suite 330,
Boston, MA 02111-1307, USA.  */

#include "config.h"
#include "system.h"

#include "rtl.h"
#include "regs.h"
#include "function.h"
#include "hard-reg-set.h"
#include "flags.h"
#include "insn-config.h"
#include "recog.h"
#include "basic-block.h"
#include "output.h"
#include "except.h"

static int identify_call_return_value    PARAMS ((rtx, rtx *, rtx *));
static rtx skip_copy_to_return_value     PARAMS ((rtx, rtx, rtx));
static rtx skip_use_of_return_value      PARAMS ((rtx, enum rtx_code));
static rtx skip_stack_adjustment         PARAMS ((rtx));
static rtx skip_jump_insn                PARAMS ((rtx));
static int uses_addressof                PARAMS ((rtx));
static int sequence_uses_addressof       PARAMS ((rtx));
static void purge_reg_equiv_notes        PARAMS ((void));

/* Examine a CALL_PLACEHOLDER pattern and determine where the call's
   return value is located.  P_HARD_RETURN receives the hard register
```

```
   that the function used; P_SOFT_RETURN receives the pseudo register
   that the sequence used.  Return non-zero if the values were located.  */

static int
identify_call_return_value (cp, p_hard_return, p_soft_return)
     rtx cp;
     rtx *p_hard_return, *p_soft_return;
{
  rtx insn, set, hard, soft;

  /* Search forward through the "normal" call sequence to the CALL insn.  */
  insn = XEXP (cp, 0);
  while (GET_CODE (insn) != CALL_INSN)
    insn = NEXT_INSN (insn);

  /* Assume the pattern is (set (dest) (call ...)), or that the first
     member of a parallel is.  This is the hard return register used
     by the function.  */
  if (GET_CODE (PATTERN (insn)) == SET
      && GET_CODE (SET_SRC (PATTERN (insn))) == CALL)
    hard = SET_DEST (PATTERN (insn));
  else if (GET_CODE (PATTERN (insn)) == PARALLEL
                && GET_CODE (XVECEXP (PATTERN (insn), 0, 0)) == SET
                && GET_CODE (SET_SRC (XVECEXP (PATTERN (insn), 0, 0))) == CALL)
    hard = SET_DEST (XVECEXP (PATTERN (insn), 0, 0));
  else
    return 0;

  /* If we didn't get a single hard register (e.g. a parallel), give up.  */
  if (GET_CODE (hard) != REG)
    return 0;

  /* If there's nothing after, there's no soft return value.  */
  insn = NEXT_INSN (insn);
  if (! insn)
    return 0;

  /* We're looking for a source of the hard return register.  */
  set = single_set (insn);
  if (! set || SET_SRC (set) != hard)
    return 0;

  soft = SET_DEST (set);
  insn = NEXT_INSN (insn);

  /* Allow this first destination to be copied to a second register,
     as might happen if the first register wasn't the particular pseudo
     we'd been expecting.  */
  if (insn
      && (set = single_set (insn)) != NULL_RTX
      && SET_SRC (set) == soft)
    {
      soft = SET_DEST (set);
      insn = NEXT_INSN (insn);
    }

  /* Don't fool with anything but pseudo registers.  */
  if (GET_CODE (soft) != REG || REGNO (soft) < FIRST_PSEUDO_REGISTER)
    return 0;

  /* This value must not be modified before the end of the sequence.  */
  if (reg_set_between_p (soft, insn, NULL_RTX))
    return 0;

  *p_hard_return = hard;
  *p_soft_return = soft;

  return 1;
}

/* If the first real insn after ORIG_INSN copies to this function's
   return value from RETVAL, then return the insn which performs the
   copy.  Otherwise return ORIG_INSN.  */

static rtx
skip_copy_to_return_value (orig_insn, hardret, softret)
     rtx orig_insn;
```

```
     rtx hardret, softret;
{
  rtx insn, set = NULL_RTX;

  insn = next_nonnote_insn (orig_insn);
  if (! insn)
    return orig_insn;

  set = single_set (insn);
  if (! set)
    return orig_insn;

  /* The destination must be the same as the called function's return
     value to ensure that any return value is put in the same place by the
     current function and the function we're calling.

     Further, the source must be the same as the pseudo into which the
     called function's return value was copied.  Otherwise we're returning
     some other value.  */

#ifndef OUTGOING_REGNO
#define OUTGOING_REGNO(N) (N)
#endif

  if (SET_DEST (set) == current_function_return_rtx
      && REG_P (SET_DEST (set))
      && OUTGOING_REGNO (REGNO (SET_DEST (set))) == REGNO (hardret)
      && SET_SRC (set) == softret)
    return insn;

  /* It did not look like a copy of the return value, so return the
     same insn we were passed.  */
  return orig_insn;
}

/* If the first real insn after ORIG_INSN is a CODE of this function's return
   value, return insn.  Otherwise return ORIG_INSN.  */

static rtx
skip_use_of_return_value (orig_insn, code)
     rtx orig_insn;
     enum rtx_code code;
{
  rtx insn;

  insn = next_nonnote_insn (orig_insn);

  if (insn
      && GET_CODE (insn) == INSN
      && GET_CODE (PATTERN (insn)) == code
      && (XEXP (PATTERN (insn), 0) == current_function_return_rtx
                || XEXP (PATTERN (insn), 0) == const0_rtx))
    return insn;

  return orig_insn;
}

/* If the first real insn after ORIG_INSN adjusts the stack pointer
   by a constant, return the insn with the stack pointer adjustment.
   Otherwise return ORIG_INSN.  */

static rtx
skip_stack_adjustment (orig_insn)
     rtx orig_insn;
{
  rtx insn, set = NULL_RTX;

  insn = next_nonnote_insn (orig_insn);

  if (insn)
    set = single_set (insn);

  /* The source must be the same as the current function's return value to
     ensure that any return value is put in the same place by the current
     function and the function we're calling.   The destination register
     must be a pseudo.  */
  if (insn
```

```
      && set
      && GET_CODE (SET_SRC (set)) == PLUS
      && XEXP (SET_SRC (set), 0) == stack_pointer_rtx
      && GET_CODE (XEXP (SET_SRC (set), 1)) == CONST_INT
      && SET_DEST (set) == stack_pointer_rtx)
    return insn;

  /* It did not look like a copy of the return value, so return the
     same insn we were passed.  */
  return orig_insn;
}

/* If the first real insn after ORIG_INSN is a jump, return the JUMP_INSN.
   Otherwise return ORIG_INSN.  */

static rtx
skip_jump_insn (orig_insn)
     rtx orig_insn;
{
  rtx insn;

  insn = next_nonnote_insn (orig_insn);

  if (insn
      && GET_CODE (insn) == JUMP_INSN
      && simplejump_p (insn))
    return insn;

  return orig_insn;
}

/* Scan the rtx X for an ADDRESSOF expressions.  Return nonzero if an ADDRESSOF
   expresion is found, else return zero.  */

static int
uses_addressof (x)
     rtx x;
{
  RTX_CODE code;
  int i, j;
  const char *fmt;

  if (x == NULL_RTX)
    return 0;

  code = GET_CODE (x);

  if (code == ADDRESSOF)
    return 1;

  /* Scan all subexpressions. */
  fmt = GET_RTX_FORMAT (code);
  for (i = 0; i < GET_RTX_LENGTH (code); i++, fmt++)
    {
      if (*fmt == 'e')
	{
	  if (uses_addressof (XEXP (x, i)))
	    return 1;
	}
      else if (*fmt == 'E')
	{
	  for (j = 0; j < XVECLEN (x, i); j++)
	    if (uses_addressof (XVECEXP (x, i, j)))
	      return 1;
	}
    }
  return 0;
}

/* Scan the sequence of insns in SEQ to see if any have an ADDRESSOF
   rtl expression.  If an ADDRESSOF expression is found, return nonzero,
   else return zero.

   This function handles CALL_PLACEHOLDERs which contain multiple sequences
   of insns.  */

static int
```

```
sequence_uses_addressof (seq)
     rtx seq;
{
  rtx insn;

  for (insn = seq; insn; insn = NEXT_INSN (insn))
    if (GET_RTX_CLASS (GET_CODE (insn)) == 'i')
      {
		/* If this is a CALL_PLACEHOLDER, then recursively call ourselves
		   with each nonempty sequence attached to the CALL_PLACEHOLDER.  */
		if (GET_CODE (insn) == CALL_INSN
		    && GET_CODE (PATTERN (insn)) == CALL_PLACEHOLDER)
		  {
		    if (XEXP (PATTERN (insn), 0) != NULL_RTX
			      && sequence_uses_addressof (XEXP (PATTERN (insn), 0)))
		      return 1;
		    if (XEXP (PATTERN (insn), 1) != NULL_RTX
			      && sequence_uses_addressof (XEXP (PATTERN (insn), 1)))
		      return 1;
		    if (XEXP (PATTERN (insn), 2) != NULL_RTX
			      && sequence_uses_addressof (XEXP (PATTERN (insn), 2)))
		      return 1;
		  }
		else if (uses_addressof (PATTERN (insn))
			       || (REG_NOTES (insn) && uses_addressof (REG_NOTES (insn))))
		  return 1;
      }
  return 0;
}

/* Remove all REG_EQUIV notes found in the insn chain.  */

static void
purge_reg_equiv_notes ()
{
  rtx insn;

  for (insn = get_insns (); insn; insn = NEXT_INSN (insn))
    {
      while (1)
		{
		  rtx note = find_reg_note (insn, REG_EQUIV, 0);
		  if (note)
		    {
		      /* Remove the note and keep looking at the notes for
			       this insn.  */
		      remove_note (insn, note);
		      continue;
		    }
		  break;
		}
    }
}

/* Replace the CALL_PLACEHOLDER with one of its children.  INSN should be
   the CALL_PLACEHOLDER insn; USE tells which child to use.  */

void
replace_call_placeholder (insn, use)
     rtx insn;
     sibcall_use_t use;
{
  if (use == sibcall_use_tail_recursion)
    emit_insns_before (XEXP (PATTERN (insn), 2), insn);
  else if (use == sibcall_use_sibcall)
    emit_insns_before (XEXP (PATTERN (insn), 1), insn);
  else if (use == sibcall_use_normal)
    emit_insns_before (XEXP (PATTERN (insn), 0), insn);
  else
    abort();

  /* Turn off LABEL_PRESERVE_P for the tail recursion label if it
     exists.  We only had to set it long enough to keep the jump
     pass above from deleting it as unused.  */
  if (XEXP (PATTERN (insn), 3))
    LABEL_PRESERVE_P (XEXP (PATTERN (insn), 3)) = 0;
```

```
  /* "Delete" the placeholder insn. */
  PUT_CODE (insn, NOTE);
  NOTE_SOURCE_FILE (insn) = 0;
  NOTE_LINE_NUMBER (insn) = NOTE_INSN_DELETED;
}

/* Given a (possibly empty) set of potential sibling or tail recursion call
   sites, determine if optimization is possible.

   Potential sibling or tail recursion calls are marked with CALL_PLACEHOLDER
   insns.  The CALL_PLACEHOLDER insn holds chains of insns to implement a
   normal call, sibling call or tail recursive call.

   Replace the CALL_PLACEHOLDER with an appropriate insn chain.  */

void
optimize_sibling_and_tail_recursive_calls ()
{
  rtx insn, insns;
  basic_block alternate_exit = EXIT_BLOCK_PTR;
  int current_function_uses_addressof;
  int successful_sibling_call = 0;
  int replaced_call_placeholder = 0;
  edge e;

  insns = get_insns ();

  /* We do not perform these calls when flag_exceptions is true, so this
     is probably a NOP at the current time.  However, we may want to support
     sibling and tail recursion optimizations in the future, so let's plan
     ahead and find all the EH labels.  */
  find_exception_handler_labels ();

  /* Run a jump optimization pass to clean up the CFG.  We primarily want
     this to thread jumps so that it is obvious which blocks jump to the
     epilouge.  */
  jump_optimize_minimal (insns);

  /* We need cfg information to determine which blocks are succeeded
     only by the epilogue.  */
  find_basic_blocks (insns, max_reg_num (), 0);
  cleanup_cfg (insns);

  /* If there are no basic blocks, then there is nothing to do.  */
  if (n_basic_blocks == 0)
    return;

  /* Find the exit block.

     It is possible that we have blocks which can reach the exit block
     directly.  However, most of the time a block will jump (or fall into)
     N_BASIC_BLOCKS - 1, which in turn falls into the exit block.  */
  for (e = EXIT_BLOCK_PTR->pred;
       e && alternate_exit == EXIT_BLOCK_PTR;
       e = e->pred_next)
    {
      rtx insn;

      if (e->dest != EXIT_BLOCK_PTR || e->succ_next != NULL)
	continue;

      /* Walk forwards through the last normal block and see if it
	 does nothing except fall into the exit block.  */
      for (insn = BLOCK_HEAD (n_basic_blocks - 1);
	   insn;
	   insn = NEXT_INSN (insn))
	{
	  /* This should only happen once, at the start of this block.  */
	  if (GET_CODE (insn) == CODE_LABEL)
	    continue;

	  if (GET_CODE (insn) == NOTE)
	    continue;

	  if (GET_CODE (insn) == INSN
	      && GET_CODE (PATTERN (insn)) == USE)
	    continue;
```

```
            break;
          }

      /* If INSN is zero, then the search walked all the way through the
             block without hitting anything interesting.  This block is a
             valid alternate exit block.  */
      if (insn == NULL)
            alternate_exit = e->src;
    }

  /* If the function uses ADDRESSOF, we can't (easily) determine
     at this point if the value will end up on the stack.  */
  current_function_uses_addressof = sequence_uses_addressof (insns);

  /* Walk the insn chain and find any CALL_PLACEHOLDER insns.  We need to
     select one of the insn sequences attached to each CALL_PLACEHOLDER.

     The different sequences represent different ways to implement the call,
     ie, tail recursion, sibling call or normal call.

     Since we do not create nested CALL_PLACEHOLDERs, the scan
     continues with the insn that was after a replaced CALL_PLACEHOLDER;
     we don't rescan the replacement insns.  */
  for (insn = insns; insn; insn = NEXT_INSN (insn))
    {
      if (GET_CODE (insn) == CALL_INSN
              && GET_CODE (PATTERN (insn)) == CALL_PLACEHOLDER)
            {
              int sibcall = (XEXP (PATTERN (insn), 1) != NULL_RTX);
              int tailrecursion = (XEXP (PATTERN (insn), 2) != NULL_RTX);
              basic_block succ_block, call_block;
              rtx temp, hardret, softret;

              /* We must be careful with stack slots which are live at
                 potential optimization sites.

                 ?!? This test is overly conservative and will be replaced.  */
              if (frame_offset)
                goto failure;

              /* alloca (until we have stack slot life analysis) inhibits
                 sibling call optimizations, but not tail recursion.

                 Similarly if we have ADDRESSOF expressions.

                 Similarly if we use varargs or stdarg since they implicitly
                 may take the address of an argument.  */
              if (current_function_calls_alloca || current_function_uses_addressof
                  || current_function_varargs || current_function_stdarg)
                sibcall = 0;

              call_block = BLOCK_FOR_INSN (insn);

              /* If the block has more than one successor, then we can not
                 perform sibcall or tail recursion optimizations.  */
              if (call_block->succ == NULL
                  || call_block->succ->succ_next != NULL)
                goto failure;

              /* If the single successor is not the exit block, then we can not
                 perform sibcall or tail recursion optimizations.

                 Note that this test combined with the previous is sufficient
                 to prevent tail call optimization in the presense of active
                 exception handlers.  */
              succ_block = call_block->succ->dest;
              if (succ_block != EXIT_BLOCK_PTR && succ_block != alternate_exit)
                goto failure;

              /* If the call was the end of the block, then we're OK.  */
              temp = insn;
              if (temp == call_block->end)
                goto success;

              /* Skip over copying from the call's return value pseudo into
                 this function's hard return register.  */
```

```
                if (identify_call_return_value (PATTERN (insn), &hardret, &softret))
                  {
                    temp = skip_copy_to_return_value (temp, hardret, softret);
                    if (temp == call_block->end)
                      goto success;
                  }

                /* Skip any stack adjustment.  */
                temp = skip_stack_adjustment (temp);
                if (temp == call_block->end)
                  goto success;

                /* Skip over a CLOBBER of the return value (as a hard reg).  */
                temp = skip_use_of_return_value (temp, CLOBBER);
                if (temp == call_block->end)
                  goto success;

                /* Skip over a USE of the return value (as a hard reg).  */
                temp = skip_use_of_return_value (temp, USE);
                if (temp == call_block->end)
                  goto success;

                /* Skip over the JUMP_INSN at the end of the block.  */
                temp = skip_jump_insn (temp);
                if (GET_CODE (temp) == NOTE)
                  temp = next_nonnote_insn (temp);
                if (temp == call_block->end)
                  goto success;

                /* There are operations at the end of the block which we must
                   execute after returning from the function call.  So this call
                   can not be optimized.  */
failure:
                sibcall = 0, tailrecursion = 0;
success:

                /* Select a set of insns to implement the call and emit them.
                   Tail recursion is the most efficient, so select it over
                   a tail/sibling call.  */

                if (sibcall)
                  successful_sibling_call = 1;
                replaced_call_placeholder = 1;
                replace_call_placeholder (insn,
                                                          tailrecursion != 0
                                                            ? sibcall_use_tail_recursion
                                                            : sibcall != 0
                                                                    ? sibcall_use_sibcall
                                                                    : sibcall_use_normal);

              }
        }

  /* A sibling call sequence invalidates any REG_EQUIV notes made for
     this function's incoming arguments.

     At the start of RTL generation we know the only REG_EQUIV notes
     in the rtl chain are those for incoming arguments, so we can safely
     flush any REG_EQUIV note.

     This is (slight) overkill.  We could keep track of the highest argument
     we clobber and be more selective in removing notes, but it does not
     seem to be worth the effort.  */
  if (successful_sibling_call)
    purge_reg_equiv_notes ();

  /* There may have been NOTE_INSN_BLOCK_{BEGIN,END} notes in the
     CALL_PLACEHOLDER alternatives that we didn't emit.  Rebuild the
     lexical block tree to correspond to the notes that still exist.  */
  if (replaced_call_placeholder)
    reorder_blocks ();

  /* This information will be invalid after inline expansion.  Kill it now.  */
  free_basic_block_vars (0);
}
```

C Appendix. A GCC Patch

This is an example of how a GCC patch could look like. This patch is constructed to be able to use the Cygnus patch on the Sparc architecture. When a developer wants to improve a program, he (she) takes the original source code and creates a copy of it. Implement the changes and write the necessary comments. When the implementation is finished, one performs an act called "diff". The diff command creates a new file, a patch that consists of pairs of code pieces and corresponding comments. A pair consist of old code and new code. Now that a patch has been created, one can distribute the patch. This relatively small sized file is easier to distribute than a complete source file. A user can apply the patch on a file by specifying the files names and write the "patch" command in a shell-tool.

When looking at a patch, some rows begin with '+++' and '---', as in the following example.

```
--- gcc/config/sparc/sparc.h.jj Fri Mar 24 09:13:57 2000
+++ gcc/config/sparc/sparc.h    Fri Mar 24 09:17:10 2000


@@ -131,7 +131,7 @@
 ;; Attributes for instruction and branch scheduling

 (define_attr "in_call_delay" "false,true"
-  (cond [(eq_attr "type"
"uncond_branch,branch,call,call_no_delay_slot,return,multi")
+  (cond [(eq_attr "type"
"uncond_branch,branch,call,sibcall,call_no_delay_slot,return,multi")
                (const_string "false")
         (eq_attr "type" "load,fpload,store,fpstore")
                (if_then_else (eq_attr "length" "1")
```

The'**@@**' tells where in the file the changes will take place and the '`;;`' tells that the row is a comment. The '---' tells which file that will be patched and the '+++' tells the new name of the file with the new timestamp. A single'-' means that this expression will be replaced with the expression that follows a single '+' as shown above.

It is not easy, even for a very experienced eye, to interpret a patch because the patch consists only of fragments of the original file, taken out of its context. In order to fully understand a patch, one has to know and understand the original file's code. The complete Sparc patch will be presented in section C.1.

C.1 Mr Jelinek's Sparc Patch

```
--- gcc/config/sparc/sparc.h.jj Fri Mar 24 09:13:57 2000
+++ gcc/config/sparc/sparc.h    Fri Mar 24 09:17:10 2000
@@ -1066,8 +1066,8 @@ do
\
           %fp, but output it as %i7.  */                      \
        fixed_regs[31] = 1;                                     \
        reg_names[FRAME_POINTER_REGNUM] = "%i7";                \
-       /* ??? This is a hack to disable leaf functions.  */    \
-       global_regs[7] = 1;                                     \
+       /* Disable leaf functions */                            \
+       bzero (sparc_leaf_regs, FIRST_PSEUDO_REGISTER);         \
       }                                                                \
     if (profile_block_flag)                                    \
       {                                                                \
@@ -1373,26 +1373,8 @@ extern enum reg_class sparc_regno_reg_cl

 #define ORDER_REGS_FOR_LOCAL_ALLOC order_regs_for_local_alloc ()

-/* ??? %g7 is not a leaf register to effectively #undef LEAF_REGISTERS when
-   -mflat is used.  Function only_leaf_regs_used will return 0 if a global
-   register is used and is not permitted in a leaf function.  We make %g7
-   a global reg if -mflat and voila.  Since %g7 is a system register and is
-   fixed it won't be used by gcc anyway.  */
-
-#define LEAF_REGISTERS \
-{ 1, 1, 1, 1, 1, 1, 1, 0,      \
-  0, 0, 0, 0, 0, 0, 1, 0,      \
-  0, 0, 0, 0, 0, 0, 0, 0,      \
-  1, 1, 1, 1, 1, 1, 0, 1,      \
-  1, 1, 1, 1, 1, 1, 1, 1,      \
-  1, 1, 1, 1, 1, 1, 1, 1,      \
-  1, 1, 1, 1, 1, 1, 1, 1,      \
-  1, 1, 1, 1, 1, 1, 1, 1,      \
-  1, 1, 1, 1, 1, 1, 1, 1,      \
-  1, 1, 1, 1, 1, 1, 1, 1,      \
-  1, 1, 1, 1, 1, 1, 1, 1,      \
-  1, 1, 1, 1, 1, 1, 1, 1,      \
-  1, 1, 1, 1, 1}
+extern char sparc_leaf_regs[];
+#define LEAF_REGISTERS sparc_leaf_regs

 extern char leaf_reg_remap[];
 #define LEAF_REG_REMAP(REGNO) (leaf_reg_remap[REGNO])
@@ -2144,6 +2126,10 @@ LFLGRET"ID":\n\
    For the v9 we want NAMED to mean what it says it means.  */

 #define STRICT_ARGUMENT_NAMING TARGET_V9
+
+/* We do not allow sibling calls if -mflat, nor
+   we do not allow indirect calls to be optimized into sibling calls.  */
+#define FUNCTION_OK_FOR_SIBCALL(DECL) (DECL && ! TARGET_FLAT)

 /* Generate RTL to flush the register windows so as to make arbitrary frames
    available.  */
--- gcc/config/sparc/sparc.md.jj        Mon Mar 13 18:05:46 2000
+++ gcc/config/sparc/sparc.md   Fri Mar 24 09:17:10 2000
@@ -88,7 +88,7 @@
 ;; type "call_no_delay_slot" is a call followed by an unimp instruction.

 (define_attr "type"
-
"move,unary,binary,compare,load,sload,store,ialu,shift,uncond_branch,branch,call,ca
```

```
ll_no_delay _slot,return,address,imul,fpload,fpstore,fp,fpmove,fpc-
move,fpcmp,fpmul,fpdivs,fpdivd,fpsqrts,fpsqrtd,cmove,multi,misc"
+
"move,unary,binary,compare,load,sload,store,ialu,shift,uncond_branch,branch,call,si
bcall,call_no _delay_slot,return,address,imul,fpload,fpstore,fp,fpmove,fpc-
move,fpcmp,fpmul,fpdivs,fpdivd,fpsqrts,fpsqrtd,cmove,multi,misc"
   (const_string "binary"))

 ;; Set true if insn uses call-clobbered intermediate register.
@@ -131,7 +131,7 @@
 ;; Attributes for instruction and branch scheduling

 (define_attr "in_call_delay" "false,true"
-  (cond [(eq_attr "type"
"uncond_branch,branch,call,call_no_delay_slot,return,multi")
+  (cond [(eq_attr "type"
"uncond_branch,branch,call,sibcall,call_no_delay_slot,return,multi")
                (const_string "false")
         (eq_attr "type" "load,fpload,store,fpstore")
                (if_then_else (eq_attr "length" "1")
@@ -148,6 +148,12 @@
 (define_delay (eq_attr "type" "call")
   [(eq_attr "in_call_delay" "true") (nil) (nil)])

+(define_attr "eligible_for_sibcall_delay" "false,true"
+  (symbol_ref "eligible_for_sibcall_delay(insn)"))
+
+(define_delay (eq_attr "type" "sibcall")
+  [(eq_attr "eligible_for_sibcall_delay" "true") (nil) (nil)])
+
 (define_attr "leaf_function" "false,true"
   (const (symbol_ref "current_function_uses_only_leaf_regs")))

@@ -179,19 +185,19 @@
 ;; because it prevents us from moving back the final store of inner loops.

 (define_attr "in_branch_delay" "false,true"
-  (if_then_else (and (eq_attr "type"
"!uncond_branch,branch,call,call_no_delay_slot,multi")
+  (if_then_else (and (eq_attr "type" "!uncond_branch,branch,call,sib-
call,call_no_delay_slot,multi")
                     (eq_attr "length" "1"))
                (const_string "true")
                (const_string "false")))

 (define_attr "in_uncond_branch_delay" "false,true"
-  (if_then_else (and (eq_attr "type"
"!uncond_branch,branch,call,call_no_delay_slot,multi")
+  (if_then_else (and (eq_attr "type" "!uncond_branch,branch,call,sib-
call,call_no_delay_slot,multi")
                     (eq_attr "length" "1"))
                (const_string "true")
                (const_string "false")))

 (define_attr "in_annul_branch_delay" "false,true"
-  (if_then_else (and (eq_attr "type"
"!uncond_branch,branch,call,call_no_delay_slot,multi")
+  (if_then_else (and (eq_attr "type" "!uncond_branch,branch,call,sib-
call,call_no_delay_slot,multi")
                     (eq_attr "length" "1"))
                (const_string "true")
                (const_string "false")))
@@ -453,7 +459,7 @@

 (define_function_unit "ieuN" 2 0
   (and (eq_attr "cpu" "ultrasparc")
-    (eq_attr "type" "ialu,binary,move,unary,shift,com-
pare,call,call_no_delay_slot,uncond_branch"))
```

```
+    (eq_attr "type" "ialu,binary,move,unary,shift,compare,call,sib-
call,call_no_delay_slot,uncond_branch"))
   1 1)

 (define_function_unit "ieu0" 1 0
@@ -468,7 +474,7 @@

 (define_function_unit "ieu1" 1 0
   (and (eq_attr "cpu" "ultrasparc")
-    (eq_attr "type" "compare,call,call_no_delay_slot,uncond_branch"))
+    (eq_attr "type" "compare,call,sibcall,call_no_delay_slot,uncond_branch"))
   1 1)

 (define_function_unit "cti" 1 0
@@ -8569,6 +8575,59 @@

   DONE;
 }")
+
+;;- tail calls
+(define_expand "sibcall"
+  [(parallel [(call (match_operand 0 "call_operand" "") (const_int 0))
+             (return)])]
+  ""
+  "")
+
+(define_insn "*sibcall_symbolic_sp32"
+  [(call (mem:SI (match_operand:SI 0 "symbolic_operand" "s"))
+        (match_operand 1 "" ""))
+   (return)]
+  "! TARGET_PTR64"
+  "* return output_sibcall(insn, operands[0]);"
+  [(set_attr "type" "sibcall")])
+
+(define_insn "*sibcall_symbolic_sp64"
+  [(call (mem:SI (match_operand:DI 0 "symbolic_operand" "s"))
+        (match_operand 1 "" ""))
+   (return)]
+  "TARGET_PTR64"
+  "* return output_sibcall(insn, operands[0]);"
+  [(set_attr "type" "sibcall")])
+
+(define_expand "sibcall_value"
+  [(parallel [(set (match_operand 0 "register_operand" "=rf")
+               (call (match_operand:SI 1 "" "") (const_int 0)))
+             (return)])]
+  ""
+  "")
+
+(define_insn "*sibcall_value_symbolic_sp32"
+  [(set (match_operand 0 "" "=rf")
+       (call (mem:SI (match_operand:SI 1 "symbolic_operand" "s"))
+             (match_operand 2 "" "")))
+   (return)]
+  "! TARGET_PTR64"
+  "* return output_sibcall(insn, operands[1]);"
+  [(set_attr "type" "sibcall")])
+
+(define_insn "*sibcall_value_symbolic_sp64"
+  [(set (match_operand 0 "" "")
+       (call (mem:SI (match_operand:DI 1 "symbolic_operand" "s"))
+             (match_operand 2 "" "")))
+   (return)]
+  "TARGET_PTR64"
+  "* return output_sibcall(insn, operands[1]);"
+  [(set_attr "type" "sibcall")])
+
+(define_expand "sibcall_epilogue"
```

```
+  [(const_int 0)]
+  ""
+  "DONE;")

 ;; UNSPEC_VOLATILE is considered to use and clobber all hard registers and
 ;; all of memory.  This blocks insns from being moved across this point.
--- gcc/config/sparc/sparc-protos.h.jj  Thu Feb 17 16:31:05 2000
+++ gcc/config/sparc/sparc-protos.h     Fri Mar 24 09:17:10 2000
@@ -96,6 +96,7 @@ extern int sparc_splitdi_legitimate PARA
 extern int sparc_absnegfloat_split_legitimate PARAMS ((rtx, rtx));
 extern char *output_cbranch PARAMS ((rtx, int, int, int, int, rtx));
 extern const char *output_return PARAMS ((rtx *));
+extern const char *output_sibcall PARAMS ((rtx, rtx));
 extern char *output_v9branch PARAMS ((rtx, int, int, int, int, int, rtx));
 extern void emit_v9_brxx_insn PARAMS ((enum rtx_code, rtx, rtx));
 extern void output_double_int PARAMS ((FILE *, rtx));
@@ -121,6 +122,7 @@ extern int cc_arithopn PARAMS ((rtx, enu
 extern int data_segment_operand PARAMS ((rtx, enum machine_mode));
 extern int eligible_for_epilogue_delay PARAMS ((rtx, int));
 extern int eligible_for_return_delay PARAMS ((rtx));
+extern int eligible_for_sibcall_delay PARAMS ((rtx));
 extern int emit_move_sequence PARAMS ((rtx, enum machine_mode));
 extern int extend_op PARAMS ((rtx, enum machine_mode));
 extern int fcc_reg_operand PARAMS ((rtx, enum machine_mode));
--- gcc/config/sparc/sparc.c.jj Wed Mar 22 08:49:19 2000
+++ gcc/config/sparc/sparc.c    Fri Mar 24 09:17:11 2000
@@ -99,6 +99,24 @@ char leaf_reg_remap[] =
   88, 89, 90, 91, 92, 93, 94, 95,
   96, 97, 98, 99, 100};

+/* Vector, indexed by hard register number, which contains 1
+   for a register that is allowable in a candidate for leaf
+   function treatment.  */
+char sparc_leaf_regs[] =
+{ 1, 1, 1, 1, 1, 1, 1, 1,
+  0, 0, 0, 0, 0, 0, 1, 0,
+  0, 0, 0, 0, 0, 0, 0, 0,
+  1, 1, 1, 1, 1, 1, 0, 1,
+  1, 1, 1, 1, 1, 1, 1, 1,
+  1, 1, 1, 1, 1, 1, 1, 1,
+  1, 1, 1, 1, 1, 1, 1, 1,
+  1, 1, 1, 1, 1, 1, 1, 1,
+  1, 1, 1, 1, 1, 1, 1, 1,
+  1, 1, 1, 1, 1, 1, 1, 1,
+  1, 1, 1, 1, 1, 1, 1, 1,
+  1, 1, 1, 1, 1, 1, 1, 1,
+  1, 1, 1, 1, 1};
+
 #endif

 /* Name of where we pretend to think the frame pointer points.
@@ -2458,6 +2476,98 @@ eligible_for_epilogue_delay (trial, slot
   return 0;
 }

+/* Return nonzero if TRIAL can go into the sibling call
+   delay slot.  */
+
+int
+eligible_for_sibcall_delay (trial)
+     rtx trial;
+{
+  rtx pat, src;
+
+  if (GET_CODE (trial) != INSN || GET_CODE (PATTERN (trial)) != SET)
+    return 0;
+
+  if (get_attr_length (trial) != 1 || profile_block_flag == 2)
```

```
+    return 0;
+
+  pat = PATTERN (trial);
+
+  if (current_function_uses_only_leaf_regs)
+    {
+      /* If the tail call is done using the call instruction,
+        we have to restore %o7 in the delay slot.  */
+      if (TARGET_ARCH64 && ! TARGET_CM_MEDLOW)
+       return 0;
+
+      /* %g1 is used to build the function address */
+      if (reg_mentioned_p (gen_rtx_REG (Pmode, 1), pat))
+       return 0;
+
+      return 1;
+    }
+
+  /* Otherwise, only operations which can be done in tandem with
+     a 'restore' insn can go into the delay slot.  */
+  if (GET_CODE (SET_DEST (pat)) != REG
+      || REGNO (SET_DEST (pat)) < 24
+      || REGNO (SET_DEST (pat)) >= 32)
+    return 0;
+
+  /* If it mentions %o7, it can't go in, because sibcall will clobber it
+     in most cases.  */
+  if (reg_mentioned_p (gen_rtx_REG (Pmode, 15), pat))
+    return 0;
+
+  src = SET_SRC (pat);
+
+  if (arith_operand (src, GET_MODE (src)))
+    {
+      if (TARGET_ARCH64)
+        return GET_MODE_SIZE (GET_MODE (src)) <= GET_MODE_SIZE (DImode);
+      else
+        return GET_MODE_SIZE (GET_MODE (src)) <= GET_MODE_SIZE (SImode);
+    }
+
+  else if (arith_double_operand (src, GET_MODE (src)))
+    return GET_MODE_SIZE (GET_MODE (src)) <= GET_MODE_SIZE (DImode);
+
+  else if (! TARGET_FPU && restore_operand (SET_DEST (pat), SFmode)
+          && register_operand (src, SFmode))
+    return 1;
+
+  else if (GET_CODE (src) == PLUS
+          && arith_operand (XEXP (src, 0), SImode)
+          && arith_operand (XEXP (src, 1), SImode)
+          && (register_operand (XEXP (src, 0), SImode)
+              || register_operand (XEXP (src, 1), SImode)))
+    return 1;
+
+  else if (GET_CODE (src) == PLUS
+          && arith_double_operand (XEXP (src, 0), DImode)
+          && arith_double_operand (XEXP (src, 1), DImode)
+          && (register_operand (XEXP (src, 0), DImode)
+              || register_operand (XEXP (src, 1), DImode)))
+    return 1;
+
+  else if (GET_CODE (src) == LO_SUM
+          && ! TARGET_CM_MEDMID
+          && ((register_operand (XEXP (src, 0), SImode)
+               && immediate_operand (XEXP (src, 1), SImode))
+              || (TARGET_ARCH64
+                  && register_operand (XEXP (src, 0), DImode)
+                  && immediate_operand (XEXP (src, 1), DImode))))
```

```
+    return 1;
+
+  else if (GET_CODE (src) == ASHIFT
+          && (register_operand (XEXP (src, 0), SImode)
+              || register_operand (XEXP (src, 0), DImode))
+          && XEXP (src, 1) == const1_rtx)
+    return 1;
+
+  return 0;
+}
+
 static int
 check_return_regs (x)
      rtx x;
@@ -3423,6 +3533,40 @@ output_function_prologue (file, size, le
     }
 }

+/* Output code to restore any call saved registers.  */
+
+static void
+output_restore_regs (file, leaf_function)
+     FILE *file;
+     int leaf_function;
+{
+  int offset, n_regs;
+  const char *base;
+
+  offset = -apparent_fsize + frame_base_offset;
+  if (offset < -4096 || offset + num_gfregs * 4 > 4096 - 8 /*double*/)
+    {
+      build_big_number (file, offset, "%g1");
+      fprintf (file, "add%s, %%g1, %%g1\n", frame_base_name);
+      base = "%g1";
+      offset = 0;
+    }
+  else
+    {
+      base = frame_base_name;
+    }
+
+  n_regs = 0;
+  if (TARGET_EPILOGUE && ! leaf_function)
+    /* ??? Originally saved regs 0-15 here.  */
+    n_regs = restore_regs (file, 0, 8, base, offset, 0);
+  else if (leaf_function)
+    /* ??? Originally saved regs 0-31 here.  */
+    n_regs = restore_regs (file, 0, 8, base, offset, 0);
+  if (TARGET_EPILOGUE)
+    restore_regs (file, 32, TARGET_V9 ? 96 : 64, base, offset, n_regs);
+}
+
 /* Output code for the function epilogue.  */

 void
@@ -3457,35 +3601,8 @@ output_function_epilogue (file, size, le
       goto output_vectors;
     }

-  /* Restore any call saved registers.  */
   if (num_gfregs)
-    {
-      int offset, n_regs;
-      const char *base;
-
-      offset = -apparent_fsize + frame_base_offset;
-      if (offset < -4096 || offset + num_gfregs * 4 > 4096 - 8 /*double*/)
-       {
```

```
-	  build_big_number (file, offset, "%g1");
-	  fprintf (file, "add%s, %%g1, %%g1\n", frame_base_name);
-	  base = "%g1";
-	  offset = 0;
-	}
-      else
-	{
-	  base = frame_base_name;
-	}
-
-      n_regs = 0;
-      if (TARGET_EPILOGUE && ! leaf_function)
-	/* ??? Originally saved regs 0-15 here.  */
-	n_regs = restore_regs (file, 0, 8, base, offset, 0);
-      else if (leaf_function)
-	/* ??? Originally saved regs 0-31 here.  */
-	n_regs = restore_regs (file, 0, 8, base, offset, 0);
-      if (TARGET_EPILOGUE)
-	restore_regs (file, 32, TARGET_V9 ? 96 : 64, base, offset, n_regs);
-    }
+    output_restore_regs (file, leaf_function);

   /* Work out how to skip the caller's unimp instruction if required.  */
   if (leaf_function)
@@ -3575,6 +3692,139 @@ output_function_epilogue (file, size, le
  output_vectors:
   sparc_output_deferred_case_vectors ();
 }
+
+/* Output a sibling call.  */
+
+const char *
+output_sibcall (insn, call_operand)
+     rtx insn, call_operand;
+{
+  int leaf_regs = current_function_uses_only_leaf_regs;
+  rtx operands[3];
+  int delay_slot = dbr_sequence_length () > 0;
+
+  if (num_gfregs)
+    {
+      /* Call to restore global regs might clobber
+	 the delay slot. Instead of checking for this
+	 output the delay slot now.  */
+      if (delay_slot)
+	{
+	  rtx delay = NEXT_INSN (insn);
+
+	  if (! delay)
+	    abort ();
+
+	  final_scan_insn (delay, asm_out_file, 1, 0, 1);
+	  PATTERN (delay) = gen_blockage ();
+	  INSN_CODE (delay) = -1;
+	  delay_slot = 0;
+	}
+      output_restore_regs (asm_out_file, leaf_regs);
+    }
+
+  operands[0] = call_operand;
+
+  if (leaf_regs)
+    {
+      int spare_slot = (TARGET_ARCH32 || TARGET_CM_MEDLOW);
+      int size = 0;
+
+      if ((actual_fsize || ! spare_slot) && delay_slot)
+	{
```

```
+          rtx delay = NEXT_INSN (insn);
+
+          if (! delay)
+            abort ();
+
+          final_scan_insn (delay, asm_out_file, 1, 0, 1);
+          PATTERN (delay) = gen_blockage ();
+          INSN_CODE (delay) = -1;
+          delay_slot = 0;
+        }
+      if (actual_fsize)
+        {
+          if (actual_fsize <= 4096)
+            size = actual_fsize;
+          else if (actual_fsize <= 8192)
+            {
+              fputs ("sub%sp, -4096, %sp\n", asm_out_file);
+              size = actual_fsize - 4096;
+            }
+          else if ((actual_fsize & 0x3ff) == 0)
+            fprintf (asm_out_file,
+                     "sethi%%hi(%d), %%g1\nadd%%sp, %%g1, %%sp\n",
+                     actual_fsize);
+          else
+            {
+              fprintf (asm_out_file,
+                       "sethi%%hi(%d), %%g1\nor%%g1, %%lo(%d), %%g1\n",
+                       actual_fsize, actual_fsize);
+              fputs ("add%%sp, %%g1, %%sp\n", asm_out_file);
+            }
+        }
+      if (spare_slot)
+        {
+          output_asm_insn ("sethi%%hi(%a0), %%g1", operands);
+          output_asm_insn ("jmpl%%g1 + %%lo(%a0), %%g0", operands);
+          if (size)
+            fprintf (asm_out_file, " sub%%sp, -%d, %%sp\n", size);
+          else if (! delay_slot)
+            fputs (" nop\n", asm_out_file);
+        }
+      else
+        {
+          if (size)
+            fprintf (asm_out_file, "sub%%sp, -%d, %%sp\n", size);
+          output_asm_insn ("mov%%o7, %%g1", operands);
+          output_asm_insn ("call%a0, 0", operands);
+          output_asm_insn (" mov%%g1, %%o7", operands);
+        }
+      return "";
+    }
+
+  output_asm_insn ("call%a0, 0", operands);
+  if (delay_slot)
+    {
+      rtx delay = NEXT_INSN (insn), pat;
+
+      if (! delay)
+       abort ();
+
+      pat = PATTERN (delay);
+      if (GET_CODE (pat) != SET)
+       abort ();
+
+      operands[0] = SET_DEST (pat);
+      pat = SET_SRC (pat);
+      switch (GET_CODE (pat))
+        {
+       case PLUS:
```

```
+         operands[1] = XEXP (pat, 0);
+         operands[2] = XEXP (pat, 1);
+         output_asm_insn (" restore %r1, %2, %Y0", operands);
+         break;
+       case LO_SUM:
+         operands[1] = XEXP (pat, 0);
+         operands[2] = XEXP (pat, 1);
+         output_asm_insn (" restore %r1, %%lo(%a2), %Y0", operands);
+         break;
+       case ASHIFT:
+         operands[1] = XEXP (pat, 0);
+         output_asm_insn (" restore %r1, %r1, %Y0", operands);
+         break;
+       default:
+         operands[1] = pat;
+         output_asm_insn (" restore %%g0, %1, %Y0", operands);
+         break;
+       }
+      PATTERN (delay) = gen_blockage ();
+      INSN_CODE (delay) = -1;
+    }
+  else
+    fputs (" restore\n", asm_out_file);
+  return "";
+}

 /* Functions for handling argument passing.

@@ -7014,6 +7264,7 @@ ultra_code_from_mask (type_mask)
     return IEU0;
   else if (type_mask & (TMASK (TYPE_COMPARE) |
                         TMASK (TYPE_CALL) |
+                        TMASK (TYPE_SIBCALL) |
                         TMASK (TYPE_UNCOND_BRANCH)))
     return IEU1;
   else if (type_mask & (TMASK (TYPE_IALU) | TMASK (TYPE_BINARY) |
@@ -7486,6 +7737,7 @@ ultrasparc_sched_reorder (dump, sched_ve
        /* If we are not in the process of emptying out the pipe, try to
           obtain an instruction which must be the first in it's group.  */
        ip = ultra_find_type ((TMASK (TYPE_CALL) |
+                              TMASK (TYPE_SIBCALL) |
                               TMASK (TYPE_CALL_NO_DELAY_SLOT) |
                               TMASK (TYPE_UNCOND_BRANCH)),
                              ready, this_insn);
--- gcc/tm.texi.jj      Fri Mar 24 09:13:52 2000
+++ gcc/tm.texi Fri Mar 24 09:17:11 2000
@@ -1652,7 +1652,7 @@ accomplish this.
 @table @code
 @findex LEAF_REGISTERS
 @item LEAF_REGISTERS
-A C initializer for a vector, indexed by hard register number, which
+Name of a char vector, indexed by hard register number, which
 contains 1 for a register that is allowable in a candidate for leaf
 function treatment.

--- gcc/sibcall.c.jj    Sun Mar 19 06:26:47 2000
+++ gcc/sibcall.c       Fri Mar 24 09:17:11 2000
@@ -140,9 +140,13 @@ skip_copy_to_return_value (orig_insn, ha
      called function's return value was copied.  Otherwise we're returning
      some other value.  */

+#ifndef OUTGOING_REGNO
+#define OUTGOING_REGNO(N) (N)
+#endif
+
   if (SET_DEST (set) == current_function_return_rtx
       && REG_P (SET_DEST (set))
-      && REGNO (SET_DEST (set)) == REGNO (hardret)
```

```
+      && OUTGOING_REGNO (REGNO (SET_DEST (set))) == REGNO (hardret)
       && SET_SRC (set) == softret)
     return insn;

@@ -352,7 +356,6 @@ replace_call_placeholder (insn, use)
   NOTE_SOURCE_FILE (insn) = 0;
   NOTE_LINE_NUMBER (insn) = NOTE_INSN_DELETED;
 }
-

 /* Given a (possibly empty) set of potential sibling or tail recursion call
    sites, determine if optimization is possible.
--- gcc/final.c.jj      Sun Mar 19 20:31:03 2000
+++ gcc/final.c Fri Mar 24 09:17:11 2000
@@ -4015,7 +4015,8 @@ leaf_function_p ()

   for (insn = get_insns (); insn; insn = NEXT_INSN (insn))
     {
-      if (GET_CODE (insn) == CALL_INSN)
+      if (GET_CODE (insn) == CALL_INSN
+         && ! SIBLING_CALL_P (insn))
        return 0;
       if (GET_CODE (insn) == INSN
          && GET_CODE (PATTERN (insn)) == SEQUENCE
@@ -4025,7 +4026,8 @@ leaf_function_p ()
     }
   for (insn = current_function_epilogue_delay_list; insn; insn = XEXP (insn, 1))
     {
-      if (GET_CODE (XEXP (insn, 0)) == CALL_INSN)
+      if (GET_CODE (XEXP (insn, 0)) == CALL_INSN
+         && ! SIBLING_CALL_P (insn))
        return 0;
       if (GET_CODE (XEXP (insn, 0)) == INSN
          && GET_CODE (PATTERN (XEXP (insn, 0))) == SEQUENCE
@@ -4048,8 +4050,6 @@ leaf_function_p ()

 #ifdef LEAF_REGISTERS

-static char permitted_reg_in_leaf_functions[] = LEAF_REGISTERS;
-
 /* Return 1 if this function uses only the registers that can be
    safely renumbered.  */

@@ -4057,6 +4057,7 @@ int
 only_leaf_regs_used ()
 {
   int i;
+  char *permitted_reg_in_leaf_functions = LEAF_REGISTERS;

   for (i = 0; i < FIRST_PSEUDO_REGISTER; i++)
     if ((regs_ever_live[i] || global_regs[i])
--- gcc/global.c.jj     Mon Mar  6 18:37:42 2000
+++ gcc/global.c        Fri Mar 24 09:17:11 2000
@@ -374,7 +374,7 @@ global_alloc (file)
      a leaf function.  */
   {
     char *cheap_regs;
-    static char leaf_regs[] = LEAF_REGISTERS;
+    char *leaf_regs = LEAF_REGISTERS;

     if (only_leaf_regs_used () && leaf_function_p ())
       cheap_regs = leaf_regs;
--- gcc/jump.c.jj       Fri Mar 24 09:13:52 2000
+++ gcc/jump.c  Fri Mar 24 09:17:11 2000
@@ -3879,6 +3879,13 @@ mark_jump_label (x, insn, cross_jump, in
                        cross_jump, in_mem);
        }
       return;
```

```
+
+  /* Look at the Normal call sequence attached to the CALL_PLACEHOLDER.  */
+    case CALL_PLACEHOLDER:
+      for (insn = XEXP (x, 0); insn; insn = NEXT_INSN (insn))
+       if (GET_RTX_CLASS (GET_CODE (insn)) == 'i')
+         mark_jump_label (PATTERN (insn), NULL_RTX, cross_jump, 0);
+      return;

     default:
       break;
--- gcc/calls.c.jj      Fri Mar 24 09:13:51 2000
+++ gcc/calls.c Fri Mar 24 09:17:11 2000
@@ -165,7 +165,7 @@ static void initialize_argument_informat
                                                         int, tree, tree,
                                                         CUMULATIVE_ARGS *,
                                                         int, rtx *, int *,
-                                                        int *, int *));
+                                                        int *, int *, int));
 static void compute_argument_addresses         PARAMS ((struct arg_data *,
                                                         rtx, int));
 static rtx rtx_for_function_call               PARAMS ((tree, tree));
@@ -980,7 +980,8 @@ static void
 initialize_argument_information (num_actuals, args, args_size, n_named_args,
                                  actparms, fndecl, args_so_far,
                                  reg_parm_stack_space, old_stack_level,
-                                 old_pending_adj, must_preallocate, is_const)
+                                 old_pending_adj, must_preallocate, is_const,
+                                 ecf_flags)
      int num_actuals ATTRIBUTE_UNUSED;
      struct arg_data *args;
      struct args_size *args_size;
@@ -993,6 +994,7 @@ initialize_argument_information (num_act
      int *old_pending_adj;
      int *must_preallocate;
      int *is_const;
+     int ecf_flags;
 {
   /* 1 if scanning parms front to back, -1 if scanning back to front.  */
   int inc;
@@ -1150,8 +1152,19 @@ initialize_argument_information (num_act

       args[i].unsignedp = unsignedp;
       args[i].mode = mode;
-      args[i].reg = FUNCTION_ARG (*args_so_far, mode, type,
-                                  argpos < n_named_args);
+
+#ifdef FUNCTION_INCOMING_ARG
+      /* If this is a sibling call and the machine has register windows, the
+        register window has to be unwinded before calling the routine, so
+        arguments have to go into the incoming registers.  */
+      if (ecf_flags & ECF_SIBCALL)
+       args[i].reg = FUNCTION_INCOMING_ARG (*args_so_far, mode, type,
+                                            argpos < n_named_args);
+      else
+#endif
+       args[i].reg = FUNCTION_ARG (*args_so_far, mode, type,
+                                    argpos < n_named_args);
+
 #ifdef FUNCTION_ARG_PARTIAL_NREGS
       if (args[i].reg)
        args[i].partial
@@ -2131,7 +2144,7 @@ expand_call (exp, target, ignore)
         call expansion.  */
       int save_pending_stack_adjust;
       rtx insns;
-      rtx before_call;
+      rtx before_call, next_arg_reg;
```

```
       if (pass == 0)
        {
@@ -2284,7 +2297,8 @@ expand_call (exp, target, ignore)
                                       n_named_args, actparms, fndecl,
                                       &args_so_far, reg_parm_stack_space,
                                       &old_stack_level, &old_pending_adj,
-                                      &must_preallocate, &is_const);
+                                      &must_preallocate, &is_const,
+                                      (pass == 0) ? ECF_SIBCALL : 0);

 #ifdef FINAL_REG_PARM_STACK_SPACE
       reg_parm_stack_space = FINAL_REG_PARM_STACK_SPACE (args_size.constant,
@@ -2305,6 +2319,13 @@ expand_call (exp, target, ignore)
          sibcall_failure = 1;
        }

+      if (args_size.constant > current_function_args_size)
+       {
+         /* If this function requires more stack slots than the current
+            function, we cannot change it into a sibling call.  */
+         sibcall_failure = 1;
+       }
+
       /* Compute the actual size of the argument block required.  The variable
         and constant sizes must be combined, the size may have to be rounded,
         and there may be a minimum required size.  When generating a sibcall
@@ -2569,9 +2590,9 @@ expand_call (exp, target, ignore)
        {
          if (pcc_struct_value)
            valreg = hard_function_value (build_pointer_type (TREE_TYPE (exp)),
-                                         fndecl, 0);
+                                         fndecl, (pass == 0));
          else
-           valreg = hard_function_value (TREE_TYPE (exp), fndecl, 0);
+           valreg = hard_function_value (TREE_TYPE (exp), fndecl, (pass == 0));
        }

       /* Precompute all register parameters.  It isn't safe to compute anything
@@ -2665,14 +2686,24 @@ expand_call (exp, target, ignore)
         later safely search backwards to find the CALL_INSN.  */
       before_call = get_last_insn ();

+      /* Set up next argument register.  For sibling calls on machines
+        with register windows this should be the incoming register.  */
+#ifdef FUNCTION_INCOMING_ARG
+      if (pass == 0)
+       next_arg_reg = FUNCTION_INCOMING_ARG (args_so_far, VOIDmode,
+                                             void_type_node, 1);
+      else
+#endif
+       next_arg_reg = FUNCTION_ARG (args_so_far, VOIDmode,
+                                    void_type_node, 1);
+
       /* All arguments and registers used for the call must be set up by
         now!  */

       /* Generate the actual call instruction.  */
       emit_call_1 (funexp, fndecl, funtype, unadjusted_args_size,
                   args_size.constant, struct_value_size,
-                  FUNCTION_ARG (args_so_far, VOIDmode, void_type_node, 1),
-                  valreg, old_inhibit_defer_pop, call_fusage,
+                  next_arg_reg, valreg, old_inhibit_defer_pop, call_fusage,
                   ((is_const ? ECF_IS_CONST : 0)
                    | (nothrow ? ECF_NOTHROW : 0)
                    | (pass == 0 ? ECF_SIBCALL : 0)));
```

D Appendix. Concepts

APZ – Real time -OS used in the AXE-system.

Backend – The backend is where the machine- and intermediate language dependent phases of the compiler belong. In the backend, most of the code optimization is performed, and machine code is generated.

Basic block – A basic block is a sequence of statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching, except at the end.

CPS – **C**ontinuation **P**assing **S**tyle. CPS is ment to resolve a longstanding structural problem in the composition of software realibility.

Cygnus – A Red Hat company. Works with software support and developement.

Epilogue – The way a function ends.

Frontend – The front end is where the source language is transformed from a high level language into a machine independent intermediate representation.

FSF – **F**ree **S**oftware **F**oundation.

GCC – **G**NU **C**ompiler **C**ollection

GNU – **G**NU's **N**ot **U**nix

Inlining – A functions code is placed directly into its caller in order to avoid an expensive function call.

Intermediate representation – Syntax trees and RTL (for GCC).

Pattern – A rule that descrives a set of strings.

RTL – **R**egister **T**ransfer **L**anguage.

Self recursive call – A kind of tail call. The function ends with a call to itself.

Sibling call – A kind of tail call when the caller and the callee has the same signature.

Signature – Two functions have the same signature when they have the same type and number of arguments.

Snapshot – An "unofficial" patched vesion of GCC, available for download inbetween two official releases.

Sparc – **S**calable **P**rocessor **ARC**itecture

Tail call – A function ends with a call to another function.

Tail call elimination – A technique used in order to reduce the run time stack usage. Do not save the caller return adress to the stack and do not save the callers local variables. Pass the caller registers to the callee.

Tail recursive call – The same as self recursive call..

TSP – Ericsson Infotech in Karlstad, Department of **T**est, **S**upport and simulated **P**latforms