TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Informatik

DIPLOMARBEIT

Übersetzung funktionaler Sprachen mittels GCC – Tail Calls

Andreas Bauer

Abgabe: 15. Januar 2003

Aufgabensteller:
Prof. Dr. Manfred Broy

Betreuer:
Dr. Markus Pizka

ERKLÄRUNG

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum Unterschrift

MUNICH UNIVERSITY OF TECHNOLOGY
Department of Informatics

A Thesis submitted for the Degree of
Diplom Informatik

Compilation of Functional Programming Languages using GCC — Tail Calls

Andreas Bauer

Date: January 15, 2003

Supervisor:
Prof. Dr. Manfred Broy

Instructor:
Dr. Markus Pizka

ACKNOWLEDGEMENTS

Thanks...

- TO my project instructors — Dr. Markus Pizka, Dr. Clem Baker-Finch, and Prof. Simon Peyton Jones: for counsel, encouragement, and for turning this thesis into an international joint venture.
- TO my project supervisor — Prof. Manfred Broy: for giving me the opportunity to work on such an interesting project.
- TO the Head of Department of Computer Science, ANU — Dr. Chris Johnson: for providing me with all the necessary facilities in an inspiring environment, again.
- TO Bob Edwards: for technical assistance and for endless personal support.
- TO Jason Ozolins: for pointing me in the right direction at the right time and for knowing the answers to basically all questions one could ever dare to ask.
- TO Fergus Henderson: for meeting me in Melbourne and providing the most valuable reviews on my source code submissions.
- TO Mark Probst: for meeting me in Munich to discuss his thesis and ideas on the subject.
- TO Richard Walker: for TEXnical assistance.
- TO the GCC developers — Richard Henderson et al.: for insightful comments on the mailing list and patience.
- TO the Canberra Linux Users Group: for advice and suggestions.
- TO my parents — Horst and Waltraud: for their unfailing love and support.

This thesis was supported by an Abroad Scholarship granted by the Deutscher Akademischer Austauschdienst (DAAD) which enabled me to finish the majority of this work at the Australian National University (ANU) in Canberra.

ABSTRACT

In the late 1980s, functional programming language implementations commonly translated a program into a C output file which could then be processed by an independent C back end. Those functional language front ends were either faced with the complexity of a single large C function that contained the functionality of the entire program, or they were constrained by the lack of support for proper tail calls in C. Even today, a lot of functional programming language implementations use C to gain native binary executables for various target platforms. Using this technique, they are still faced with the same problems as their early predecessors were; proper tail calls are still missing as a feature of modern C back ends.

This work gives the technical background and rationale as to why it is so difficult to implement proper tail call support in a system's C compiler and it discusses different approaches to overcome this traditional problem in the widespread and freely available GNU Compiler Collection (GCC), so that functional language front ends like The Glasgow Haskell Compiler (GHC), which use GCC, gain a greater amount of flexibility. It also shows how many C compiler constraints which make tail call support such a difficult feature in terms of implementation, date back to design issues made in early versions of the UNIX operating system.

In order to address, in particular, GHC's need for support of indirect tail calls in the GCC back end, a series of GCC source code changes are described which have become integral parts of the compiler suite. The changes enable the open source compiler to optimise a class of indirect tail calls on Intel-based 32 and 64-bit platforms and are designed to be portable to any other platform which is not bound by its ABI to not support this notion. A GCC test suite to verify such future ports has been developed and is described as well.

Furthermore, this work examines the GCC project in general, whose ever growing core exceeds 500,000 lines of code, and it explains how this huge open source product is being developed and driven forward by a worldwide community of programmers. In doing so, a special emphasis is placed on the process of becoming involved in the community and how to start contributing ideas and code that implements them. Because of the remarkable number of over 200 supported hardware and software platforms, this work also explains the compiler's highly portable internals in detail and it shows how GCC can be used to bootstrap a variety of different cross compilers.

CONTENTS

INTRODUCTION

The GNU Compiler Collection (GCC) offers optimised code generation for a variety of imperative and object oriented programming languages, such as Ada, C, C++, Java, Pascal, and others. Furthermore, it supports (far) more than 200 different software and hardware platforms [Pizka 1997] which strongly relates to the fact that its source code is free and open, i. e. released under the terms of the GNU General Public License [Free Software Foundation 1991]. That means, everybody can change the compiler suite's source code as long as the original copyright is not violated and the changes are published under this very same license enabling others to benefit from these changes. It is this very license, which allows others to port the collection to their platform of choice.

Today's development community around the compiler suite is large — in fact, very large. It consists of hundreds of volunteers, paid software engineers, contractors, students, government employees (e. g. NASA) and, of course, an even larger user base. A team of researchers around Prof. Simon Peyton Jones of Microsoft Research, Cambridge, UK, deploys GCC in their own project *The Glasgow Haskell Compiler* (GHC; see Peyton Jones et al. [1992] and Appendix B), a functional programming language implementation that achieves portability by using GCC as a compiler back end. The concept is simple, but very effective: GHC translates a Haskell input file into a complete C program, which can then be compiled into native assembler on every platform that GCC supports. In this way, the researchers' focus can rest solely on their Haskell front end, while an independent group of developers maintains a highly portable back end.

However, compared to imperative programming languages, the functional counterparts like Haskell, Scheme, ML, or Lisp demand fundamentally different implementation concepts. One of the most outstanding differences between the two worlds is the occurrence of a large number of *tail calls* in functional programs which, of course, also appear in the GHC generated intermediate C source code. As § 1 will explain more accurately, a tail call can be thought of as a function call which is the last instruction of the calling function.

The C notion to translate *any* function call is to allocate a block of memory on the system's run time stack, called the *stack frame*, which holds sufficient space for local variables, arguments, the return address, and so on. When functions "tailcall" to another function though, it means that their computation has basically finished at this point (as it is the *last* instruction), so the stack frame should become redundant and thus be deleted before issuing the call. This, however, is not how C commonly works or what it was designed for. In C, a

stack frame gets removed when the called function (the *callee*) returns control back to the caller and not any sooner. Hence, given a sufficiently large number of such tail calls in a C program, the run time stack would overflow, because redundant stack frames would not be released in time. (§ 1 also contains a more detailed explanation of this entanglement and the severe problems associated with it.)

For many years, people have tried to come up with a solution to the tail call optimisation problem in C, because GHC is, by far, not the only functional language implementation which relies on a C compiler's back end; amongst others there are, for example, SCHEME->C [Bartlett 1989], Mercury [Henderson et al. 1995], and Bigloo [Serrano and Weis 1995]. They all share the same concerns regarding a common C compiler's shortcoming in terms of tail call support.

The most obvious, and most frequently presented solution to the tail call problem (in GCC) would be to simply reuse stack frames and realise the function call via a jump command rather than the manufacturer's special call instruction. The advantage of such an approach is that a jump, generally, does not reserve run time stack space to create a new frame (see also § 1). In theory, this sounds very simple and straightforward but, obviously, it must be very hard to implement in a real world compiler like GCC, because after nearly two decades of existence, GCC still lacks such a very important optimisation feature.

THE AIMS OF THIS THESIS

> It all comes down to methods. Everybody wants results, but no one is willing to do what it takes to get them.
>
> — Dirty Harry, *Sudden Impact* (1983)

If the efficient translation of tail calls as they are broadly used not only by functional programs but also by their accompanying intermediate C code, is such a crucial aspect in successfully creating executable programs, the question arises, "Why has yet no one come up with a good implementation to overcome this fundamental problem?" After all, a system like SCHEME->C dates from 1989 and a decade later, the methodology as well as the availability of optimising C back ends which are able to handle tail calls *efficiently*, seems to be unchanged.

Although, over the years, many different papers, articles and theses (e. g. Nenzén and Rågård [2000], or Probst [2001]) have been written which deal with this problem as well as to provide potential answers to the above question, it was not yet possible to find a concept which would effectively enable a popular compiler suite like GCC to *properly* optimise tail calls. As a matter of fact, a lot of people still underestimate the huge number of problems involved in such a task, because they fail to see even the most basic limitations that a language like C imposes on a functional front end. This is why some of the more recent publications also examine alternative languages, such as `C--`, to represent the intermediate functional program (see, for example, Peyton Jones et al. [1999], or Pizka [2002]).

The research associated with this thesis was always meant to be a "hands on" project, meaning that its aim was to make a practical impact, rather than restate the well known results which have been concluded several times before without actually tackling the tail call problem (of GCC) at all. In an early meeting at the University of Technology in Munich, Prof. Simon Peyton Jones made it very clear that the successful compilation of functional programming languages using GCC as a back end depends *strongly* on the optimisation of tail calls, and that any results of this thesis would be useless (to his GHC project), should they fail to make practical impact. Thus, a primary goal of this work was not only to examine the problem (again), but also to remove some of the constraining factors which, so far, hindered tail call optimisation in GCC. All the attendants[1] agreed to the idea that merely rephrasing the problem would not be an option this time.

Hence, this thesis is not only meant to present an *in depth* analysis of the reasons why GCC offers poor tail call support, but it will also sketch solutions, their implementation and, finally, also code changes which have been adopted into mainline GCC to make a *noticeable* difference in terms of successful tail call optimisation.

Furthermore, this thesis fulfils yet another purpose, which is to aid as an introductory text to GCC's internals and its general development, because a lot of the common documentation targets only at experienced GCC programmers. Hence, topics like intermediate code representation, the various compilation stages and even cross compilation are covered to help others getting used to this extensive compiler suite. Also, great emphasis has been placed upon software engineering aspects, i. e. the thesis deals with questions like "How can a code base which exceeds 500,000 lines of code remain maintainable?", or "Which is the best starting point when trying to modify GCC yourself?", etc. To very experienced (open source) software developers, some of the answers may come naturally, but for most people it is quite miraculous to see a huge project, as GCC is, progress, especially because most of its developers have never met or even spoke in person.

TERMINOLOGY AND NOTATION

There are many different, partly misleading names associated with the GNU Compiler Collection. Just recently (Dec. 2002) the GCC steering committee has officially abandoned the old style name "GNU CC", referring to the GNU Compiler Collection. Nowadays, people commonly use the acronym GCC when they mean the whole suite but use, for example, GNU C, if they only refer to the C compiler in particular. This document will follow that tradition and will refer to the compiler collection as "GCC", and uses "GNU C" to refer only to its C components.

[1] The meeting consisted of Prof. Simon Peyton Jones (Microsoft Research, Cambridge, UK), Dr. Markus Pizka (University of Technology, Munich, Germany), Mark Probst (University of Technology, Vienna, Austria), and the author of this text.

References and citations in this text appear in the form § 2.5 (Section 2.5), or as Knuth [1998a, § 2.5] (Section 2.5 of Donald Knuth's book *The Art of Computer Programming*, 1998). Many authors prefer to use page numbers, instead of citing chapters and sections, but due to the fact that this document used both German and English editions of the very same books, the chosen approach seemed much more reasonable and also practical. The use of italic font in this text is mainly for emphasis (e. g. "this can be applied to *all* lines alike") and for first occurrences of important terms (e. g. "a *sibcall* is a special call mechanism").

THE STRUCTURE OF THIS THESIS

The body of the thesis is divided into seven chapters:

- § 1 deals with the basic concepts and terminology underlying this work. It accurately explains the notion of "tailcalling", the concept of using stack frames and gives reasons why a C back end may not be well suited to implement a general tail call optimisation at all.
- In § 2, the GCC project is outlined — its structure, the way people work on it, and its internals, i. e. the part which actually generates code for various targets.
- The existing handling of tail calls in GCC is described in § 3. This chapter introduces custom terminology, which is unique to GCC, and examines the technical reasons for GCC's inability to eliminate tail calls.
- § 4 discusses various different approaches to the problem and also sketches their technical realisation in the compiler suite.
- The implementation of a rather pragmatic approach which actually caused a practical impact not only on the GCC community is described in § 5 of this work.
- § 6 describes this thesis' results by giving practical examples and some important use cases of the newly achieved enhancements.
- The conclusions in § 7 describe the current state of GCC, after the solution has been adopted and elaborates on future work needed to turn the results into an even broader optimisation mechanism for the compiler.

There are four appendices: Appendix A contains basic compiler terminology which should be covered by the according text books, but may be useful as an appendix when trying to understand this text, because all the required terms can be found together in a single document. Experienced (compiler) programmers may want to skip this part. Appendix B contains Internet addresses (URLs) to software referred to in this thesis and which did not seem fit for the bibliography. The source code which the text refers to, i. e. the implementation

of the proposed solutions, can be found in Appendix C and on an attached CD-ROM. The hardware and software which was used to work on this interesting problem is summarised in Appendix D.

This document was typeset on the UNIX-like operating system GNU/Linux, using Donald Knuth's excellent document preparation system TEX together with Leslie Lamport's macro package LATEX 2_ε. During the six months of work, all text has been under version control, automated by CVS, GNU Make and various other UNIX tools. The fonts used in this document are Computer Modern, Computer Modern Sans and Computer Modern Typewriter. Unless, explicitly denoted otherwise, the figures have been either drawn manually by using TEX commands, or by using the X11-based vector drawing program Xfig by Brian V. Smith, and the interactive plotting program Gnuplot by Thomas Williams and Colin Kelley.

CHAPTER ONE

BASIC CONCEPTS

This chapter introduces the basic concepts and terminology which are underlying this work. It also presents some short examples and important applications to help make these ideas and terms transparent.

1.1 FUNCTIONAL PROGRAMMING LANGUAGES

Unlike imperative programming languages, the functional world emphasises the evaluation of expressions, rather than the execution of commands. In most cases, the mere definition of a function is already the program's source code itself. For example, a recursive definition of Euclid's algorithm [Knuth 1998b, § 4.5.2] to get the greatest common divisor of two numbers may look like this:

$$gcd(a,b) = \begin{cases} a, & \text{if } a = b \\ gcd(a-b,b), & \text{if } a > b \\ gcd(a,b-a), & \text{if } a < b \end{cases}$$

In a functional programming language like Haskell, the body of function `gcd` could then be implemented accordingly:

```
gcd a b | a==b = a
        | a>b  = gcd (a-b) b
        | a<b  = gcd a (b-a)
```

This small piece of code satisfies computation and is, indeed, very similar to the mathematical definition of the function. In fact, it looks so much alike, that it can be intuitively understood even by people who do not possess any programming skills.

What is more, functional programs consist *only* of such functions and the entire program itself is also written as a function. Generally, there are no assignment statements (which are common in any imperative language) and computation is solely based upon the arguments of functions and the corresponding return values. This notion is also known as "side effect free", because functions are not able to change anything except their return values.

1.2 TAIL CALLS

One of the most essential techniques for providing such a "natural" way of programming with values and functions is the concept of *recursion* and in particular *tail recursion*, just as it gets applied in the example of `gcd`: the recursive call to `gcd` is always the last instruction until, finally, $a = b$.

Definition (Tail Call). *A call from function f to function b is a tail call iff (in a series of instructions) the call is the last instruction of f. A tail call is tail recursive iff f and b are not two distinct functions.*

Basically, this sums up a more formal definition by Clinger [1998]. It implies that the contents of the caller's *activation record* (see Appendix A) become redundant when issuing a tail call (with disregard to certain technical aspects which may prevent that and which are discussed in § 3.3).

Example 1. The call to function `bar` is in the tail position as it is the very last instruction of `foo`. During the execution of the subroutine, the previous activation record is no longer required:

```
int foo (float a, float b)
{
  ...
  return bar (a/2);
}
```

Example 2. In this example the final call to `bar` is not a tail call, even though it seems to be in the tail position. The last instruction of `foo` is really the assignment of `bar`'s return value. The activation record of the caller must be kept alive during the execution of the subroutine. Its contents do not become redundant:

```
void foo (float a, float b)
{
  int c = 0;
  ...
  c = bar (a/2);
}
```

The second example shows that it is not always trivial to decide whether a call is really in the tail position or not. While there are efforts by people trying to detect tail calls on a syntactic level [Probst 2001, § 3], the decision is commonly pushed into a compiler's back end or, to be more precise, into the *basic block analysis* (see Appendix A).

Another case where it is not obvious at all whether the caller deals with a tail call or not, is given in the next example.

Example 3.

```
void foo (float a, float b)
{
  ...
  bar (a/2);;
}
```

The additional semicolon after the call to `bar` basically translates to a NOP instruction[1]. All good compilers realise this and will remove NOPs during optimisation and then go on to handle the cleaned up code "as always". However, with syntactic tail call detection this is not possible. Hence, this artificial example, like any other C peculiarity, would demand a specific case-by-case precaution in the parser.

Definition (Proper Tail Call). *A tail call from function f to function b is a proper tail call iff the memory held by the stack frame of f can be released before jumping to function b. A proper tail call is properly tail recursive iff f and b are not two distinct functions.*

In other words, the essence of proper tail calls is that a function can return by performing a tail call to any other function, including itself [Clinger 1998]. Naturally, functional programming languages like ML, Haskell, Scheme and many others, rely heavily on the efficiency of such tail calls. In fact, the IEEE standard for Scheme [IEEE Computer Society 1991, § 1.1] states:

> "Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

While new language implementations may be able to provide this level of integration, it is often lacking in already existing frameworks, such as the GNU Compiler Collection (GCC) being originally designed to support merely imperative programming, and therefore not dependent on recursion and tail calls, at least up to the same extent.

Example 4. A slightly modified version of the Newton Square Root (NSR) algorithm [Knuth 1998b, § 4.3.1, 4.3.3] will help to demonstrate how much the implementation of an algorithm can, indeed, depend upon proper tail calls and what happens to even simple computations, if such an optimisation is lacking. NSR is a tail recursive approximation for a value's square root. The discussed example uses an accuracy of 0.0000001 to compute the square root of 101. The according C source code is available in Appendix C.

As can be seen in Fig. 1, the NSR program aborts with a segmentation fault after approximately 28 seconds[2] during which time it used 261,953 tail calls

[1] NOP is an acronym for "No Operation".

[2] The time measured contains I/O for printing the number of function calls made. Without such debugging information, the program aborts almost instantly.

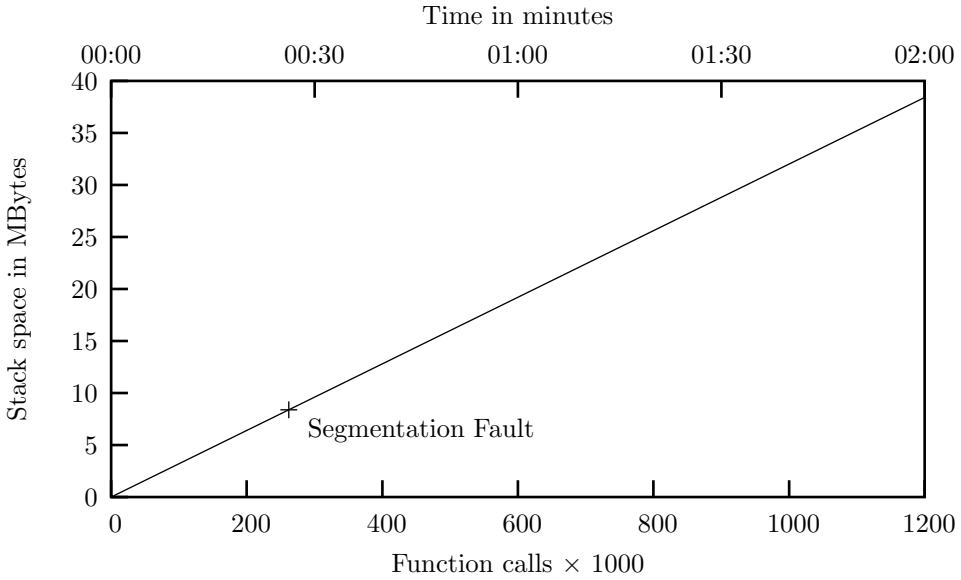


Fig. 1. Using the input value 101, the NSR program, which can be found in Appendix C, aborts after approximately 28 seconds of run time.

and 8.38 MBytes of run time stack space (on the same computer system that is described in Appendix D). The graph also shows that NSR would have consumed 40 MBytes of stack space, if it was possible to run the application for two minutes. This, however, is hard to achieve with most of today's software and hardware systems. Even though the algorithm is correct and works well for a number of input values (e. g. 100), the presented NSR implementation can hardly be used to compute the square root of (say) 101, if compiled with an unmodified GCC.

In addition to that, an ordinary function call is also rather expensive compared to a proper tail call. A call sequence of 10,000,000 ordinary calls (and, consequently, returns or the stack would overflow) required 290 ms on the same test architecture, while an implementation which deployed proper tail calls finished in less than half the time, after only 130 ms.[3]

1.3 GNU C AS INTERMEDIATE TARGET

A number of different programming languages, not only those that are functional, use GCC as an intermediate target platform, because it saves the implementors from maintaining a compiler back end for every hardware and software platform they aim to support. GCC is widespread, well maintained, and highly portable, and therefore, in many cases, ideally suited to offer a new language and associated advantages, even to exotic systems.

[3]This can easily be verified with GCC version 3.4 (or greater) which contains this work's improvements, either by modifying NSR accordingly, or by calling some "dummy" functions in a loop and, respectively, tail recursive. Time measured does *not* contain I/O.

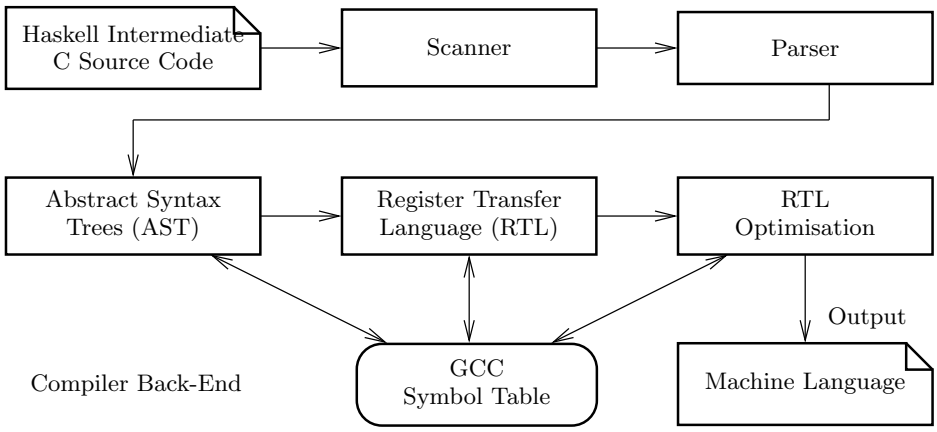


Fig. 2. The different passes of the GNU C compiler.

As already pointed out in the introduction to this text, one of these "newer" language implementations using this approach is the Glasgow Haskell Compiler (see Appendix B) which was started in 1992 by a team of researchers led by Prof. Simon Peyton Jones [Peyton Jones et al. 1992] and which also provided the original motivation for this work[4]. The Glasgow Haskell Compiler (GHC) is a complete implementation of the functional programming language Haskell, but is also able to perform as just a front end to GCC, in particular to GNU C, because it can translate the Haskell source into a C program. Figure 2 roughly sketches the process a GHC generated C source code has to go through in order to obtain assembly output for a certain architecture. RTL (Register Transfer Language) and AST (Abstract Syntax Trees) are GCC's internal intermediate code representation.

Of course, the "one big shortcoming" GHC has to deal with by using GCC as its back end is the lack of support for proper tail calls. A situation that was partly addressed by using a Perl script called "Evil Mangler"[5]. Its purpose is to make programs properly tail recursive, using pattern matching on the assembly code of function epilogues and prologues. Even though the mangler proves useful, the ultimate goal is to get rid of this script in favour of a clean and portable solution in GHC's back end. (Obviously, pattern matching on assembly is everything but portable. In fact, it can be considered "evil".)

Other front ends, especially earlier ones, have taken a different approach and compiled their language into a single large C function to overcome the problem of a growing and finally collapsing run time stack. Janus, for example, is one of these language implementations [Gudeman et al. 1992]. However, these projects share other probably even tougher, problems because of the high level of complexity found in their target C function.

[4] In comparison, the first official releases of GNU C were already available in the mid 1980s and have been maintained ever since.

[5] More in depth information on the "Evil Mangler" is available from *The Glasgow Haskell Compiler Commentary* at `http://www.cse.unsw.edu.au/~chak/haskell/ghc/comm/`.

The intermediate C code of GHC, on the other hand, is not easy to read either, although it is divided into semantically and syntactically disjoint functions. Not only to make it at least somewhat easier for the back end, GHC maintains its own internal and independent stack frame for function arguments and the like. Effectively, most subroutines do not accept ordinary parameters because there is simply no need to store them, conventionally, a second time on the machine's main run time stack. Most of the GHC generated function calls are *indirect* though, i. e. via pointers, which relates partly to the fact that GHC's execution model can be largely based upon the *Continuation Passing Style* (CPS). In CPS, the program flow (i. e. the computation) relies upon an implicit parameter, the continuation which points to the next function being executed. It is, of course, a big advantage not having to worry about function arguments too much when trying to specifically optimise GHC's back end, but the indirect calls bear great inconveniences as well. Not all platforms support the idea of indirect calls equally well, and GCC being designed for portability seems not necessarily the best choice of a compiler when extensions need to be added that primarily refuse to work on some systems (see also § 3.3).

Example. The call to the externally linked function `bar` is indirect via function pointer `ptr`. It is, in fact, an argumentless tail call and a sure candidate for optimisation:[6]

```
int (*ptr) ();       /* Function pointer.  */
extern int bar ();   /* Externally linked function.  */

int foo (int arg)
{
  ...
  ptr = bar;         /* Pointer points at bar now.  */
  return (*ptr) ();  /* Indirect tail call.  */
}
```

1.4 THE APPLICATION BINARY INTERFACE

In order to understand why GNU C does not yet offer proper tail calls, it is also important to look at the historic evolution of the compiler and its language.

In 1973, when the UNIX operating system was entirely rewritten in C, it basically meant that its core could be ported to other platforms within months [Salus 1994]. The availability of the first ports to different, but similar hardware made it soon necessary to standardise an Application Binary Interface (ABI) for each family of processors, such as Intel `ix86` for instance. Typically, an ABI

[6] Note, the example uses the "old style" notation for indirect calls, followed by *The C Programming Language* [Kernighan and Ritchie 1988, § 5.11], because the `()`-operator originally took a function and an argument list; and the function operand of `()` did not decay to a pointer. Therefore, when referring to an actual pointer, it had to be written as `(*ptr) ()`. In accordance to the ANSI-C standard [American National Standard for Information Systems 1989], modern compilers will also accept the syntax `ptr ()`.

defines a system interface for *compiled* application programs, enabling users to transfer programs from one such architecture to another. Furthermore, it describes how C types align to bytes and bits and what the memory layout generally looks like.

Stack Frames. Since the C language was originally designed for, and implemented on UNIX [Kernighan and Ritchie 1988], it comes as no surprise that the ABI was specified with this language in mind. C, being an imperative language, uses and opens new *stack frames* upon each function call (regardless whether it is in the tail position or not) whose layout is defined by the corresponding ABI. The stack frame organisation for the Intel `i386` architecture, as it was originally published together with the UNIX System V documentation [The Santa Cruz Operation 1996] by AT&T is depicted in Fig. 3.

Position	Contents	Frame	
$4n + 8$ (`%ebp`)	argument word n	Previous	*High addresses*
	...		
8 (`%ebp`)	argument word 0		
4 (`%ebp`)	return address	Current	
0 (`%ebp`)	previous `%ebp` (optional)		
-4 (`%ebp`)	unspecified		
	...		
0 (`%esp`)	variable size		*Low addresses*

Fig. 3. The standard UNIX stack frame on Intel `i386`.

Calling Conventions. The arguments for a called function, commonly referred to as the *callee*, are pushed onto the stack by the *caller* and in reverse order. That is, the syntactically first argument is pushed onto the stack last. This enables C to support functions, taking a variable amount of arguments as with, for instance, `printf`:

```
int printf (const char *format, ...)
```

On UNIX and UNIX-like systems, this convention is also known as the *C Calling Convention*. Other languages, like Pascal or Fortran use their own standards which, in general, are not compatible with C programs. In C, the arguments are usually referenced relative to a *frame pointer* (`%ebp`), while function variables are addressed via a negative offset to the *stack pointer* (`%esp`). Note, however, that a frame pointer is not always required.

Therefore, GNU C, like all other C compilers, translates a function call into a prologue sequence to create a stack frame providing space for the activation record. The called function in turn expects the compiler to put the arguments on top of its own stack frame and it expects to have space for its own variables at the expanding lower end of the stack as is required by the calling convention.

Example. In accordance with the C Calling Convention, the Intel `i386` machine code for a function call under UNIX may look like this:

```
foo:
        ...
        call    bar             Make function call (in the tail position).
        leave
        ret
bar:
        pushl   %ebp            Prologue. Push base pointer onto stack.
        movl    %esp, %ebp      Set stack pointer accordingly.
        subl    $88, %esp       Allocate stack space.
        movl    %edi, -4(%esp)  Save register.
        movl    %esi, -8(%esp)  Save register.
        ...
```

And the corresponding epilogue at the end of a function's machine code looks typically like this:

```
        ...
        movl    %edi, %eax      Set up return value.
        addl    $80, %esp       Epilogue. Free local stack space.
        popl    %edi            Restore register.
        popl    %esi            Restore register.
        leave                   Restore frame pointer.
        ret                     Pop return address.
```

Having to restore registers, as it happens in this example, is not a necessity. It really depends on the use of (which) registers, types of arguments and so on. As a matter of fact, many function calls can be translated without precautions for callee saved registers.

Table 1. INTEL `i386` REGISTERS AND THEIR ASSIGNED ROLE

Name	Role	"Belongs" to
`%eax`	Function return register; otherwise scratch.	Callee
`%ebp`	Frame pointer holds base address for stack frame; optionally, a scratch register.	Call*er*
`%ebx`, `%edi`, `%esi`	No specified role.	Call*er*
`%ecx`, `%edx`	Scratch registers.	Callee
`%esp`	Stack pointer to bottom of stack frame.	Call*er*
`%st(0)`	Floating point return register. (Must be empty upon function entry and exit.)	
`%st(1)`–`%st(7)`	Floating point scratch registers. (Must be empty upon function entry and exit.)	

The designated roles of registers in the C Calling Convention on Intel `i386` systems can be seen in Table 1. On this platform all registers are global, thus

visible to both the caller and the callee. `%ebp`, `%ebx`, `%edi`, `%esi`, and `%esp` are the *callee saved* registers, all others belong to the called function; these *caller saved* registers must be saved *before* making a call, so that their values may be preserved.

Moreover, the calling convention also defines who cleans up what parts of the run time stack. In C, it is common for the caller to reserve and clean up stack space for the callee's incoming arguments. So the callee is left with the responsibility to clean up its own local space, but *not* the argument space.

Naturally, the run time stack will grow during the execution of a C program, but usually not to the extent that the stack would "overflow" because functions release memory at the moment where control returns to the caller, i. e. old stack frames get destroyed by the epilogue. But, if applied to functional programs, this concept would prove of no value at all, since the "unusually high" level of recursion (compared to a C program) and in particular tail recursion would require too many live stack frames, hence memory, and even the most robust run time environments would asymptotically fail to satisfy such demands (see Example 4, p. 8).

Table 2. THE C AND PASCAL CALLING CONVENTIONS

Calling Convention	Argument Passing	Stack Maintenance
C	Right to left	Calling function pops arguments from the stack
Pascal	Left to right	Called function pops arguments from the stack

Due to the fact that GCC, C and UNIX are historically tied together in the sense that GCC adopted the ABIs and calling conventions for the C language in a UNIX-like environment, it now becomes clear why this popular and powerful compiler needs more than just a "few changes" in order to support a wholly new programming concept which is totally different to "trusty old C". And even though GCC is able to use a Pascal-like[7] calling convention (see Table 2), amongst others, it is often not trivial to mix and link foreign concepts together into one application or run time library. The Pascal-like calling convention uses a different epilogue, freeing space occupied by the incoming arguments:

```
bar:
        ...
        leave                   Epilogue. Restore frame pointer.
        ret     $4              Pop return address and additional 4 bytes
                                of the own argument space.
```

[7]The convention is called "`stdcall`" and was introduced rather early, due to the fact that operating systems like Apple's Mac OS and Microsoft Windows originally offered a Pascal-like system interface. It can be applied by tagging function declarations with `__attribute__ ((stdcall))`.

Another way to mix different calling conventions, other than those already supported by mainstream GCC, is opened up by the *System V Application Binary Interface* [The Santa Cruz Operation 1996] itself. The ABI requires only those functions to conform to the system's standard, which are declared in a global scope and thus can be linked by other program modules. Naturally, two external modules have to agree on a calling convention in order to be linked, but it does imply that one program alone can not make up its own internal conventions. In order to issue system calls though, such a program must still comply to the operating system's interface definition. (Of course, on UNIX-like environments it is typically a C interface, although others are also possible.)

Alternatives. Parameter passing on a system must not necessarily conform *exactly* to the presented conventions in order to stay compatible. Minor modifications are tolerated, especially if a target's hardware was designed to support them, as is the case with AMD's and Intel's 64-bit architectures, for instance. One such alternative is *Passing in Registers*. That is, a limited number of predetermined registers are used to pass a callee's arguments. Language constructs like `structs` or `unions` are still passed via the stack as it is defined by the C Calling Convention. Such a modification constitutes a consistent, therefore compatible stack frame layout and improves system performance at the same time.

1.5 THE DISEASE OF C

> Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high level languages has been a step backwards from which we may never recover.
>
> — C. A. R. Hoare, *Hints on Programming Language Design* (1973)

Yet another important reason why C compilers in general have difficulties offering proper tail calls to functional language front ends is a feature of C itself. It is perfectly legal to assign the address of a short lived function's local to a long lived *global pointer*. While this can be a "disease" when being faced with its impact, there are situations where this concept proves useful. For example, it is not possible to implement some functions such as `strtok` without a similarly high degree of flexibility in place, and even (some of its more sensible) implementors realise this by putting a warning statement in their `man` page (see Fig. 4). However, `strtok` still became part of the ANSI-C standard [American National Standard for Information Systems 1989, § 4.11.5.8] and appeared first in *The C Programming Language* [Kernighan and Ritchie 1988, § B3].

Naturally, such "pointer acrobatics" is not only counter intuitive to the functional world's notion of side effect free programming, but also makes source code hard to read and understand.

Example 1. In the following code snippet, the tail call to `bar` can not be transferred into a *proper* tail call, because the stack frame of `foo` is still alive (or, should be) when issuing control to the subroutine:

```
NAME
       strtok, strtok_r - extract tokens from strings

SYNOPSIS
       #include <string.h>

       char *strtok(char *s, const char *delim);

       char *strtok_r(char *s, const char *delim, char **ptrptr);

                                    ...

BUGS
       Never use these functions. If you do, note that:

              These functions modify their first argument.

              The identity of the delimiting character is lost.

              These functions cannot be used on constant strings.

              The  strtok()  function  uses a static buffer while
              parsing, so it's not thread safe. Use strtok_r() if
              this matters to you.
```

Fig. 4. Excerpt from the UNIX `man` page of `strtok(3)`.

```
int *global;
...
float foo ()
{
  int local = 0;
  global = &local;
  return bar (global);
}
```

Example 2. To emphasise this point and to show how "obscure" things can get in C, the code can even be made to look "worse". The problem, however, remains the same, though now it is well disguised: the pointer `global` is an *external* pointer to a pointer which in turn points to a value:

```
float foo ()
{
  extern int ***global;
  int local = 0;
  **global = &local;
  return bar (global);
}
```

Clearly, this legal example (see ISO/IEC JTC1/SC22/WG14 1999, § 3.1.2.1, for verification) is not suited for (say) pure syntactic analysis. The references, even though embedded in a function, are still global and in order to know whether the call to `bar` can be turned into a proper tail call, some in depth examination is essential.

Such a high level of indirection (in an already particularly "dangerous code") does not make sense and the majority of language front ends would be well advised to avoid it, but in order to implement a clean mechanism for detecting and optimising proper tail calls it is absolutely necessary to cover all aspects of "legal C", even if they are only of marginal practical value, as it is the case in the given examples.

As a consequence, C back ends desiring to *automatically* detect proper tail calls, must provide a sophisticated *liveness analysis*. That is, at call site, it must be known whether it is safe to scrape the locals off the stack or not, as the subroutine may still have references to contents of the caller's stack frame. The C programmer, however, does not necessarily benefit from a complicated detection algorithm, since he would be out of his mind deploying such an unsafe mechanism of referencing locals; and he is also able to manually resolve deep recursion — simply speaking — by jumping to code labels or by using loops anyway (see Aho et al. [1986, § 2.5]). In most "pure C scenarios", it does not matter whether the caller's stack frame is kept during the execution of subroutines, or not. Therefore, not too many C compiler authors would place this feature high up in their list of priorities, very much to the disadvantage of functional language front ends and the like. Potential authors may consider this as a large overhead for relatively little gain in return.

So it is probably not wrong to say that C pointers are a "blessing", because they offer flexibility beyond the possibilities of other languages like Java or Lisp for instance, and they are a "disease" at the same time, because the very concept can be wickedly abused. Even though that poses a serious problem in terms of tail call optimisation, there are different ways to overcome it. Chapters § 4 and § 5 discuss several different approaches to making compiler improvements without introducing new error potentials due to pointer constraints.

CHAPTER TWO

THE GNU COMPILER COLLECTION

The following sections introduce the GNU Compiler Collection (GCC), its internals and describe how development revolves around it. Without doubt, GCC is one of the largest and most complicated open source software packages available today, right next to giants like OpenOffice (see Appendix B) and the various UNIX-style operating system kernels [Cubranic 1999, Wheeler 2001]. And despite the helpful GCC mailing lists and online archives, it comes as no surprise that it takes quite a long while for someone to get comfortable with its inner structure and the way it is being driven forward by the worldwide community of developers. A very good understanding of these aspects is required though, when trying to contribute code to this project.

2.1 OVERVIEW

The original GNU C compiler was started in the early 1980s by Richard Stallman, who aimed to create a whole operating system based on free software. While his various tools (Bison, Emacs, etc.) that should have enabled him to create the free operating system GNU[1] grew and gained popularity, the kernel development itself stalled somewhere on the way. (This unfortunate situation was fixed a few years later when Linus Torvalds used these tools to write Linux, which then became known as GNU/Linux.)

Nowadays, it is hard to find an operating system for personal computers that is not supported by GCC, or which does not ship the compiler as an integrated part of the platform. Apple's Mac OS X (version 10.2) is compiled entirely using GCC 3.1, for instance. But also, companies like Sun, The Santa Cruz Operation (SCO), and others offer GCC as their standard system compiler. Manufacturers of handheld devices, such as Compaq with its iPAQ series that run on RISC processors of the ARM family, can use GCC to compile their application code. As a matter of fact, there are various commercial embedded Linux distributions available (e. g. Lineos's Embedix Linux) which do exactly that.

[1] GNU is a recursive acronym for "GNU's Not Unix". Trying to resolve it may corrupt the human brain.

These properties emphasise GCC's flexibility and portability alike, but also point to the fact that its code base appears to be a "monster", when daring a first glance into it. At the time of writing this thesis, the core of it exceeds 500,000 lines of code, rapidly growing as new languages, features and targets get added.

Table 1. PARTIAL LIST OF SUPPORTED TARGETS

Target	Supported Platforms	*Some* Systems and Binary Formats
`alpha`	DEC/Compaq/HP Alpha family	`-alpha-linux`, `-alpha-pc-linux-gnu`
`arm`	ARM, StrongARM family	`-elf`, `-aout`
`hppa`	HP's PA-RISC series	`-hpux`, `-linux`
`i386`	Intel 32-bit family	`-elf`, `-aout`, `-linux*aout`
`ia64`	Itanium processor family	`-linux`
`m68k`	Motorola 68k family	`-aout`, `-coff`
`mmix`	Donald Knuth's hypothetical RISC processor	`-knuth-mmixware`
`rs6000`	IBM and Motorola PowerPC family, RS/6000 systems	`-darwin`, `-linux-gnu`, `-netbsd`, `-aix`
`sparc`	Sun Sparc processors	`-netbsd-*`, `-elf`, `-sun-solaris*`
`x86_64`	AMD `x86-64`	`-linux`

*Further combinations between the types or versions of operating systems and the supported binary formats are possible.

Table 1 contains a few of the GCC supported target platforms. (It is not part of this thesis to describe them all in great detail, but to show how complex and flexible GCC's code generation design really is.) Each target description consists of a triplet which has the form `cpu-vendor-os`, where `os` can be `system` or `kernel-system`, as there are operating systems supporting more than just one standard [Vaughan et al. 2000]. For example, it is possible to compile C programs for `i386-gnu-linux-elf` and for `i386-gnu-linux-aout`, both being different targets. Sometimes, it is enough to specify merely a tuple such as `arm-elf`, `arm-linux` or `armv2-linux` for instance, with differences being only subtle. This is usually done when the `vendor` tag is unknown, or only a single vendor is supported. The number of combinations of all supported platforms and binary formats exceeds 200, including systems like Donald Knuth's `MMIX` RISC processor which does not even physically exist, except in his (upcoming rewrite of the) book series *The Art of Computer Programming* [Knuth 1998a] and in a designated virtual environment called "MMIXware" [Knuth 1999]. It is even possible to create code for nearly all of these targets without running the compiler natively. This process is called "cross compilation" and enables developers to generate foreign assembly code, for example ARM code on an Intel-based host (see § 2.3). In the case of `MMIX`, this is the *only* option to obtain code at all.

Project Management. GCC, like other large open source projects with a huge number of developers, is managed mainly via mailing lists and by a steering committee who decides upon release schedules and the general direction of the package. The source code itself is version controlled by a public CVS[2] server that only the official GCC maintainers may write to — anyone else has read-only access. All changes to the central source code repository are first presented and discussed on the lists before a maintainer either agrees to accept the contribution, or explains why he rather rejects it. In fact, even the maintainers themselves present their code changes to the public mailing lists in order to get feedback and to find out how their planned contribution interacts with different parts of the compiler suite.

Such a public review can be very useful to eliminate obvious bugs in new code, but also helps upcoming contributors getting used to the development style and cycle of GCC. Furthermore, it ensures that only high quality code finds its way into the CVS repository. The GCC project has the reputation of having one of the most strict reviewing processes of all open source projects, even more stringent than (say) the reviewing process for the Linux kernel. Then again, it can be questioned whether a bug in Linux is quite as bad as a bug in GCC, which is being used to compile Linux and the software that runs on top of it.

Maintainers also require code changes to be thoroughly tested first and to be made against the "right" CVS branch. This means that GCC improvements have to be made against the `basic-improvements-branch`, bug fixes usually against "mainline" and other changes against specific branches for a particular purpose, e. g. as it is the case with the `ssa` or the `mips-3_4-rewrite-branch`. The following list (which is far from complete) contains some CVS tags and branch names:

`gcc_latest_snapshot:` This tag marks the latest code snapshot of all of GCC. Snapshots are made daily or weekly to get a working code base that represents ongoing development and can be used for testing (e. g. SuSE is running regular SPEC2000[3] tests on development snapshots) and documentation purposes.

`gcc-3_2-branch:` This tag marks the development branch of the GCC 3.2.x series. This is where small fixes and documentation updates should be applied.

`x86-64-branch:` This branch contains a stable version of GCC for the AMD `x86-64` development/port.

`pch-branch:` This branch contains work in progress towards a precompiled header implementation, including a lot of garbage collector changes. (Many of these changes are already extant in Apple Computer's own version of GCC and are waiting to be merged into this branch.)

[2]CVS is an acronym for Concurrent Versions System (see `http://www.cvshome.org/`).

[3]The SPEC CPU2000 suite contains a number of C, C++, Fortran77 and Fortran90 programs that are compiled by GCC and then run with defined inputs (see `http://www.spec.org/osg/cpu2000/`).

`gcc-3_4-basic-improvements-branch:` This branch is for basic, straightforward improvements to the compiler. In general, this branch is away furthest from official release, as new features need more time to be tested than small bug fixes and the like.

`mips-3_4-rewrite-branch:` This branch is for "largish" rewrites to the `mips` back end.

At the time of writing, the latest stable release of GCC is version 3.2. However, a bug fix branch to the upcoming release 3.2.1 is also maintained, as well as a branch for the future release 3.3, and finally another branch for all new features that will show up for the first time in version 3.4 of the compiler (probably another two years from now). "Special branches" are usually not directly connected to any of these, but will merge the changes from time to time just to keep up.

Submitting a Patch. Contributions to GCC are commonly made by sending a *patch* to the mailing list. A patch is a piece of code which contains all the differences, found by the `diff` command of one file (or many), compared to a newer version of this file. The patch must address the correct CVS branch and it must follow certain style guidelines, set up by the GCC maintainers and accessible via the official web page. The guidelines cover aspects like the preferred indentation of code, the handling of return values and the formatting of error messages, etc. Not following the established coding conventions usually results in not getting a patch approved; a rewrite would then be necessary.

Another requirement is to test the compiler first with a new patch in place. This can be done by running the extensive internal regression test suite and by making use of GCC's many DejaGnu input files. DejaGnu is a common framework for testing programs, providing a single front end for all the individual test cases available. Like many other programs, GCC employs it to test specific features of the compiler and its supported targets. Testing is, in fact, a rather substantial part of GCC (at the time of writing, taking up about 20% of the total space of the `basic-improvements` CVS tree) with companies like Red Hat paying engineers to do (almost) nothing else, but trying to make GCC's testing procedures more perfect. Interestingly enough, the tests have gotten so complex that they can hardly be pursued by a single person anymore. For example, Red Hat runs daily regression tests on numerous different platforms while, at the same time, monitoring GCC's `ChangeLog` very carefully. If a new regression is detected, no matter on which platform, the bug gets associated with a `ChangeLog` entry and the author of the according patch is receiving an email with a description of the problem.

Since an automated framework can not cover all possible faults and traps, it is also absolutely essential to *bootstrap* (see Appendix A) the compiler before publishing code changes to it. Should the changes affect multiple platforms at once, bootstrapping has to be performed at least with a cross compiler or, alternatively, native.

However, not all patches address the code base of GCC. It is also possible to submit changes to other parts of the project, like documentation or its web page. But even then it is required to test these changes and to make sure that everything (e. g. the documentation) "builds". The GCC manual is written in the Texinfo format [Chassell and Stallman 1999] and integrates smoothly into its overall build process. The input files can also be used to compile the documentation into Postscript, PDF, HTML, or LaTeX format.

Copyright Assignment Process. GCC, like all other "GNU Software" is released under the General Public License [Free Software Foundation 1991]. That is, GCC's source code is free and open to anyone interested, including commercial vendors, but must not be modified without applying this very same license to all changes. One of the implications of that is that, every contributor has to write to the Free Software Foundation (FSF), which is currently holding the copyright on GCC, and apply for a "Copyright Assignment Form". If the FSF has received the signed form from the potential contributor, thereby signifying an agreement to accept the terms of the GPL and that the contributed code is free from any rights of a third party, it is possible for program changes to be considered for submission.

For the sake of writing this thesis, such a form has been signed by the author and by the University of Technology, Munich, to assign the copyright of all of the author's changes back to the FSF.

This "tedious" step is absolutely necessary to ensure that all code submissions can be accepted without having to worry about a university or an employer claiming copyright on them several years later when it is already too late to remove the affected code.

Parties Involved. Although GCC's copyright is owned by the FSF, many other parties with commercial interests pay their staff to help develop the software or to maintain servers to host it. Of course, it implies that they have read, understood and accepted the terms of the GPL, just like anyone else.

Some of these parties involved, for example, are companies like Red Hat and SuSE who ship the compiler with their Linux distribution, Apple who base their entire system software on GCC, IBM who use GCC for system development on a wide range of platforms and many, many others. It is no coincidence that most of the GCC maintainers are being paid for their work by some of these companies. They are professional software engineers (professional in a sense that they are no less skilled than developers working on "closed source" software) continuously enhancing the code quality and stability of the compiler suite. And they are also the reason for the strict code reviewing process each submission has to undergo. There are, however, countless numbers of private people and students working on GCC as well, who are not being paid by anyone and are merely doing it to finish a course project or to gain experience in developing open source software.

Problems. But not all is well in the world of GCC; sheer project size and numerous language front ends bear risks and problems, especially in terms of maintenance. While GCC grew over the years, the number of people with an overview over the package as a whole, has shrunk. Nowadays, it is mainly the 14 members from the steering committee who know how all the different components of GCC interact and are connected with each other (contrary to the popular belief that open source software is always and necessarily developed by thousands or millions of keen volunteers, at the same time).

All bigger projects have similar maintenance concerns, but not all have these concerns resolved by the same methods. GCC's approach to these problems is both a public and pragmatic one including the following cornerstones:

- It has an up to date *online documentation* which reflects all the latest developments; it is available via the official GCC homepage and contains useful information to developers and users alike. Most of the content is contributed by developers though, because all program changes that may affect the API and also internal interfaces, must be documented, promptly. The documentation is, just like the source code itself, under version control and it is not uncommon for patches on documentation to get rejected on the mailing lists, e. g. due to bad formatting or use of language.
- Program changes must be documented by *commenting the source code* to ensure that other developers understand the rationale of the changes. Accepted patches must also be accompanied by a detailed `ChangeLog` entry which states the author and the general purpose of the patches. Some claim that GCC's best documentation is its source code and the comments; and this, in many ways, is true.
- The GCC project has several *mailing lists:* `gcc-help` to discuss matters ranging from a failing program installation up to questions about copyright assignments and the like, `gcc` to discuss the general development of the GNU Compiler Collection, including its front ends, `gcc-patches` to discuss new source code and documentation submissions, `gcc-bugs` to report problems with the compiler or its code generation, etc. These lists are a formidable entry point for GCC development, and their extensive, searchable online archives give answers (at least) for common questions almost instantly.
- A chief concept of GCC is *modularity*, also to enable program ports with relatively little effort. Unlike other, sometimes faster compilers, GCC's architecture is layered and structured and patches trying to implement a new feature at the expense of modularity are almost certainly rejected to keep the project as "clean" as possible (for a detailed description of the layers and stages of the compiler suite, see § 2.2). After all, GCC does not impose deadlines or has "feature hungry" clients who possess the economic power to demand the latest compiler properties at any cost if necessary.

With these and the other aforementioned techniques in place it is and has been quite possible to manage the "GCC monster" successfully over the last two decades. As the compiler suite expands, new CVS branches will be created and concurrent versions of code and documentation kept, but despite the breathtaking size of the project, development will not cease at all. The modular concept of GCC does probably not even demand more than a steering committee to

understand the "bigger picture" and it is quite possible even for beginners to become GCC experts by starting out with modifying only certain bits of it, not knowing the details of neighbouring parts.

Although all this is good news, it still takes skilled developers and software engineers who are capable of applying these techniques and tools, such as CVS, to a comprehensive and distributed project like GCC; because, even the most sophisticated project management would be worthless, if there were no experts able to apply it in practice. For example, it takes usually more than a week, before people are allowed to add further patches to the repository again, when the GCC project merges the mainline CVS trunk with a development branch — obviously, not a trivial task at all.

2.2 INTERNALS

According to Richard Stallman, GCC's original goal "was to make a good, fast compiler for machines in the class that the GNU system aims to run on: 32-bit machines that address 8-bit bytes and have several general registers. Elegance, theoretical power and simplicity are only secondary. [Stallman 2002]" Nowadays, the compiler runs on many other systems than just 32-bit and keeps getting ported as new architectures hit the markets (e. g. AMD's `x86-64`).

The major reason for GCC being greatly portable is that it contains no machine specific code. However, it does contain code that "depends on machine parameters such as endianness (whether the most significant byte has the highest or lowest address of the bytes in a word) and the availability of autoincrement addressing [Stallman 2002]."

It was a second, implicit goal of the GCC project to generate high quality code for all different target platforms through intensive optimisation. This goal could only be equally achieved on all supported platforms alike by performing the most crucial parts of the optimisation in a target-independent manner. Hence came the necessity for RTL (Register Transfer Language), which is GCC's main intermediate code representation, in important parts independent of the processor GCC is running on. The architecture for the RTL machine is an abstraction of actual processor architectures. It therefore reflects the technical reality of familiar concepts like storage, processor registers, address types, an abstract jump command, and so on. In RTL it is possible to express almost all of the input program, though not to the extent that would allow front ends to use it solely as an interface. RTL is meant for internal use only, and modifying GCC's functionality almost always results in modifying the way it handles and generates RTL on a particular host, or for a particular target.

For those situations in which it is useful to examine the internal RTL representation of a function, GCC is able to produce a dump from within a certain stage of compilation, e. g. right before the sibcall optimisation pass starts. The RTL dumps are syntactically similar to the Lisp programming language, though not directly related (see Fig. 1).

The proper way to interface GCC to a (new) language front end is with the `tree`, or AST (Abstract Syntax Tree) data structure, described in the source

```
(insn 11 10 12 (nil) (set (reg/f:SI 59)
        (mem/f:SI (symbol_ref:SI ("ptr")) [0 ptr7+0 S4 A32])) -1 (nil)
    (nil))

(call_insn 12 11 14 (nil) (set (reg:SI 0 eax)
        (call (mem:QI (reg/f:SI 59) [0 S1 A8])
            (const_int 4 [0x4]))) -1 (nil)
    (nil)
    (nil))
```

Fig. 1. Part of a typical RTL dump; two instructions (`insns`), preparing an indirect function call via the pointer `ptr`.

files `tree.h` and `tree.def`. They are the foundations of RTL generation and often it is necessary to have multiple strategies for one particular kind of syntax tree, that are usable for different combinations of parameters. Such strategies may also vary greatly from target to target, but can be addressed in a machine-independent fashion, and will affect only the platforms that really need adjustments.

2.2.1 Passes of the Compiler

Figure 2 shows a simplified diagram of GCC's compilation stages of which only some are relevant for this work. In order to keep things in perspective, only the important passes are described in this chapter. For a more complete explanation that covers the whole process in detail, see the *GNU Compiler Collection Internals* [Stallman 2002].

Parsing. This pass reads the entire text of a function definition, constructing a high level (Abstract Syntax) tree representation. Because of the semantic analysis that takes place during this pass, it does more than is formally considered to be parsing. Also it does not entirely follow C syntax in order to support a wider range of programming languages. The important files (for this work at least) of this pass are `tree.h` and `tree.def` which define the format of the tree representation, `c-parse.in` which describes the main parser itself, and a few more which will appear later in this text.

RTL Generation. The conversion of the Abstract Syntax Tree data structure `tree` into RTL code takes place in this pass. It is actually done statement-by-statement during parsing, but for most purposes it can be thought of as a separate single pass. This is also where the bulk of target-*parameter*-dependent code is found and where optimisation is done for `if` conditions that are comparisons, boolean operations or conditional expressions. Most importantly though, tail calls and recursion are detected at this time. Furthermore, decisions are made about how best to arrange loops and how to output `switch` statements. Among others, the source files for RTL generation include `stmt.c`, `calls.c`, `expr.c`, `function.c` and `emit-rtl.c`. Also, the file `insn-emit.c`, generated

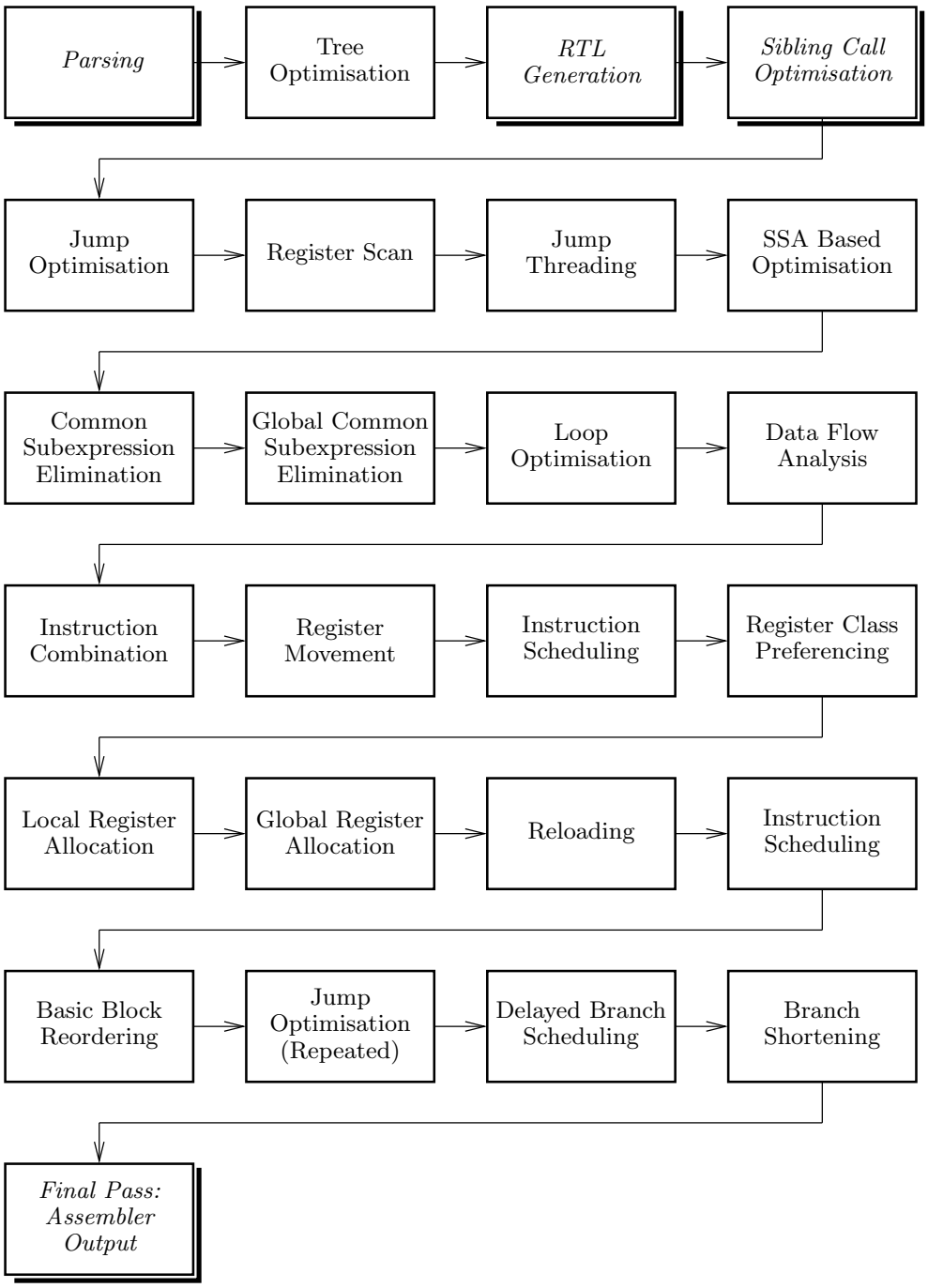


Fig. 2. A simplified, sequential diagram of the compilation process in the back end, when a sufficiently high level of optimisation is enabled.

from a target machine description is deployed in this pass. The header file `expr.h` is used for "communication" within this stage.

Sibling Call Optimisation. An accurate definition of a *sibling call* can be found in § 3. This pass is responsible for tail recursion elimination, and tail and sibling call optimisation. The purpose of these optimisations is to reduce the overhead of function calls, whenever possible. A main source file of this pass is `sibcall.c`, even though a lot of decisions on optimisation candidates are made in earlier stages (even more so due to the outcome of this work).

Final Pass (Assembler Output). In this pass, the assembly instructions for a function are created and machine-specific peephole optimisation is performed at the same time. The function entry and exit sequences are produced directly as assembler code, i. e. they never exist in RTL. The interesting source files of this pass are `final.c` and `insn-output.c`. The latter is generated automatically from the machine description by a tool called "`genoutput`". The header file `conditions.h` is used for "communication" between these files.

2.2.2 Abstract Syntax Trees

"The central data structure used by the internal representation is the `tree`. These nodes, while all of the C type `tree`, are of many varieties. A `tree` is a pointer type, but the object to which it points may be of a variety of types [Stallman 2002]." In many ways the trees[4] correspond to ordinary syntax trees, e. g. as they are described by Aho et al. [1986], but they surpass their semantic expressiveness by allowing nodes to carry additional information used in later compilation stages. To give an example, there are certain tree slots reserved for the back end which can hold generated RTL, or additional front end information such as the "tailness" of a function call. (The latter does not currently exist in mainstream GCC. This thesis contains source code to implement it (see § 4.2).)

With the `TREE_CODE` macro it is possible to determine the kind of a tree node, such as `INTEGER_TYPE`, `FUNCTION_TYPE`, `POINTER_TYPE`, etc. Other macros exist to test certain attributes of tree nodes, such as `TYPE_VOLATILE_P` which checks whether a function returns at all. In general, a macro can be thought of as a predicate if it ends with "`_P`".

Figure 3 shows the usage of the `TREE_TYPE` macro and some additional fields and flags a tree node typically contains. Most of the fields are for internal use only, but `lang_flag_`x is typically reserved for language front ends and not further specified anywhere in the GCC documentation. `TREE_TYPE` can be used to obtain all sorts of different information, e. g. when applied to a pointer to a function (`POINTER_TYPE`) it returns the function type; when applied to a function type it can be used to obtain the return type, and so on.

[4] From this point forward, trees will be referred to in normal type, rather than in `this font`, except when talking about the actual C type `tree`.

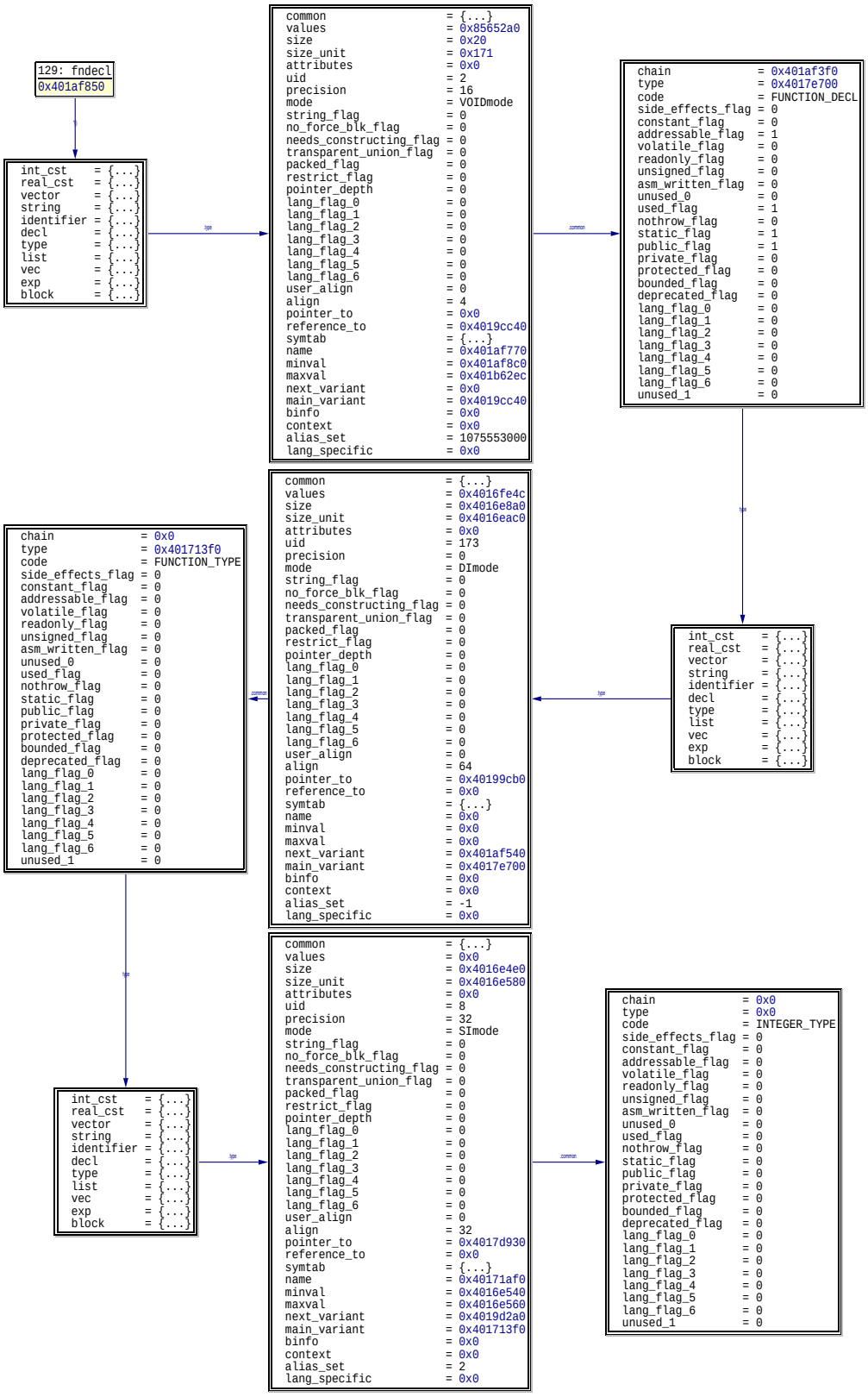


Fig. 3. The macro `TREE_TYPE` reveals a node `FUNCTION_TYPE`, containing the type of a function declaration; when applied to that, it returns `INTEGER_TYPE`, its return type. (Image created using the `gdb` debugger in combination with the `ddd` user interface (see Appendix B).)

2.2.3 Register Transfer Language

Most of the work of the compiler is done on the intermediate RTL representation. "In this language, the instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does [Stallman 2002]." It has both an internal form made up of pointers and structures, and an external, Lisp-like textual form which could also be seen in Fig. 1 (p. 25), for instance.

Example. A more extensive example of RTL code is presented here, where a function `foo` tailcalls `bar`. The following, simple C code

```
int foo ()
{
  return bar (5);
}
```

translates to this long (though not quite complete) list of RTL expressions, dumped before the actual sibling call optimisation phase took place:

```
(call_insn 19 9 20 (nil) (call_placeholder 16 10 0 0
    (call_insn 17 16 18 (nil)
    (set (reg:SI 0 eax)
           (call (mem:QI (symbol_ref:SI ("bar")) [0 S1 A8])
                  (const_int 4 [0x4]))) -1 (nil)
        (expr_list:REG_EH_REGION (const_int 0 [0x0])
            (nil))
        (nil))) -1 (nil)
    (nil)
    (nil))

(insn 20 19 21 (nil) (set (reg:SI 58)
        (reg:SI 59)) -1 (nil)
    (nil))

(jump_insn 21 20 22 (nil) (set (pc)
        (label_ref 25)) -1 (nil)
    (nil))

(barrier 22 21 23)

(note 23 22 27 NOTE_INSN_FUNCTION_END)

(insn 27 23 28 (nil) (clobber (reg/i:SI 0 eax)) -1 (nil)
    (nil))

(insn 28 27 25 (nil) (clobber (reg:SI 58)) -1 (nil)
    (nil))
```

```
29
30  (code_label 25 28 26 6 "" [0 uses])
31
32  (insn 26 25 29 (nil) (set (reg/i:SI 0 eax)
33          (reg:SI 58)) -1 (nil)
34      (nil))
35
36  (insn 29 26 0 (nil) (use (reg/i:SI 0 eax)) -1 (nil)
37      (nil))
```

The trailing numbers of an RTL instruction indicate which place in the intermediate language list is occupied by it (e. g. the second last `insn` is 26, line 32), followed by two further numbers (in this case, 25 and 29) that indicate to which objects it is linked to. The remaining characters are the instruction itself; register `%eax` is assigned a value of type integer. (The "Single Integer" mode `SI` represents a four-byte integer.) Its origin is an artificial *pseudo register* (58), because the compiler's strategy is to generate code assuming an unlimited number of such pseudo registers, and later convert them into hard registers or even into memory references. The many "`-1 (nil)`" references mean that the compiler has not yet matched this part of RTL to the possibilities given in the according *machine description*. Later on, these gaps are filled and instructions are rearranged. Further examples and RTL descriptions are available in GCC's Internals documentation [Stallman 2002] or in *Porting the GNU C Compiler to the Thor Microprocessor* by Gunnarsson et al. [1995].

RTL uses five kinds of objects: expressions, integers, wide integers, strings and vectors. Expressions are the most important ones. An RTL expression ("RTX" for short) is a C structure, but it is usually referred to with a pointer; a type that is given the `typedef` name `rtx`. Expressions are classified by expression codes, also called "RTX codes". The expression code is a name defined in the file `rtl.def`, which is also (in upper case) a C enumeration constant. The possible expression codes and their meanings are machine-independent. The code of an RTX can be extracted with the macro `GET_CODE` and altered with `PUT_CODE` [Stallman 2002]. For instance, there are RTX codes that represent an actual object, such as a register, or a memory location as well as codes for comparison, such as `NE` ("not equal") or `LT` ("lower than"), or codes for general arithmetic, and so on.

2.2.4 Machine Descriptions

A machine description has at least three parts: a file of instruction patterns (`.md` file), a C header file of macro definitions and a C source which contains target-specific hooks and the like. The `.md` file for a target machine contains a pattern for each instruction that the target machine supports (or at least each instruction that is worth telling the compiler about). The header file conveys information about a target that does not fit the scheme of the `.md` file and the main C source file specifies further capabilities of the hardware which *should not* be expressed purely by the instruction patterns.

When the compiler builds, it really interprets the machine description for a target to create further C source files. These are compiled and linked with the rest of GCC's sources to create an executable binary. By offering such a large variety of machine descriptions, it now becomes clear why GCC is suited so well for cross compilation. Technically, all it takes to get a foreign compiler is to configure GCC with an alternative machine description in place and then recompile. (§ 2.3 gives details to that.)

Instruction Patterns. The contents of the `.md` file are a mixture of RTL and C syntax to define and fill instruction patterns with semantics, so the compiler is able to use them for a certain target. The patterns either start with `define_insn` or `define_expand`. The latter is used only when converting the parse trees into RTL, i. e. into RTXs which, in turn, are defined by the `define_insn` declarations.

The following instruction pattern is a `define_insn` taken from `i386.md`; it defines an indirect function call to a `void` function:

```
(define_insn "*call_1"
 [(call (mem:QI (match_operand:SI 0 "call_insn_operand" "rsm"))
        (match_operand 1 "" ""))]
 "!SIBLING_CALL_P (insn) && !TARGET_64BIT"
{
  if (constant_call_address_operand (operands[0], QImode))
    return "call\t%P0";
  return "call\t%A0";
}
  [(set_attr "type" "call")])
```

The name of this pattern is "`*call_1`" and any instruction whose RTL description has the form shown, satisfying the *condition* "`!SIBLING_CALL_P && !TARGET_64BIT`" may be handled according to this pattern. The description contains a template (`match_operand`) which is used to define which instructions match the particular pattern and how to find their operands. The operands, in turn, are restricted to a *predicate* (`call_insn_operand`) and further *constraints* (`rsm`). The predicate is the name of a C function which accepts two parameters, an expression and a machine mode. The constraint is usually a class of registers or memory locations to use for an operation. `rsm` stands for "register-, immediate-integer-, and memory-operands allowed". Finally, the output control string is a piece of C code which chooses the right output template to return, based on the kind of operand and machine mode.

Source Files. Most targets have two C files: a header, `machine.h`, and the according `machine.c` file. However, others have more to address certain portability issues like binary formats and different operating systems; for example, an additional file "`machine-linux.h`" may be present. Among other things, a header file defines memory layout and alignment issues as well as register classes and their usage. The corresponding macros are sometimes rather large

and have a function-like character that do not help the purpose of debugging much, should it be required:

```
#define MUST_PASS_IN_STACK(MODE, TYPE)                                  \
  ((TYPE) != 0                                                          \
   && (TREE_CODE (TYPE_SIZE (TYPE)) != INTEGER_CST                      \
       || TREE_ADDRESSABLE (TYPE)                                       \
       || ((MODE) == TImode)                                            \
       || ((MODE) == BLKmode                                            \
           && ! ((TYPE) != 0                                            \
                 && TREE_CODE (TYPE_SIZE (TYPE)) == INTEGER_CST         \
                 && 0 == (int_size_in_bytes (TYPE)                      \
                          % (PARM_BOUNDARY / BITS_PER_UNIT)))           \
           && (FUNCTION_ARG_PADDING (MODE, TYPE)                        \
               == (BYTES_BIG_ENDIAN ? upward : downward)))))
```

This random `i386.h` macro evaluates to non-zero if it can not be determined how to pass `TYPE` solely in registers; endianness and alignment are taken into account by deploying even further macros such as `FUNCTION_ARG_PADDING`, or `BYTES_BIG_ENDIAN`.

Depending on their return value, the C source file uses these macros to finally emit machine language. It also defines *target hooks* to optimise the output for machine-specific features, such as the availability of register windows and scratch registers, for instance. Target hooks may have a noticeable effect on the quality of the generated assembly code, if the compiler was invoked with the "right" optimisation switches. Further information on target hooks is also available in § 5.

2.3 Building the Compilers

Although there exists good and extensive documentation on how to build GCC on various platforms, it can be somewhat "tricky" at times, especially when trying to do something non-standard like *cross compilation*. This section is devoted to show the issues involved[5] and gives reason for the importance of cross compilation when working on GCC's code base.

After downloading either a CVS version or a compressed archive of GCC, the user has to invoke an enclosed `configure` script for guessing and also setting system-specific values, such as where the current C compiler resides and which libraries are available, etc. This information is required by the build process to compile and later install GCC on a certain platform. While this process is fairly standard in the GNU/UNIX world, the vast amount of configuration switches

[5]The alert reader might be wondering at this point, why a process like software installation is addressed in a thesis, but he should be aware of the fact the author spent a great part of his restricted time in trying to figure out good ways of setting up cross compilers to test his various GCC changes on different platforms. The importance of this step and the experiences gained should not be underestimated.

available is still somewhat uncommon. Albeit, it is enough to call `configure` with a single option, in order to set up purely the C compiler on and for the current host:

```
$ ./configure --enable-languages=c
```

It does get more complicated when trying to build a cross compiler, as several additional programs and libraries need configuration and building first. GCC does not *fully* compile without access to a variety of standard system calls, on GNU-based systems typically provided by the GNU C library (`glibc`). In fact, the whole process can become barely possible on older machines, as stated in *The PARISC-Linux Cross Compiler HOWTO*[6]:

> "You are going to need a good and fast cross compiling box and at least 1.5 GBs of free drive space to fit all the source and compiled stages.
>
> You will also need all the latest tools required to build: `glibc`, `gcc`, `binutils` and the latest Linux kernel."

Generally speaking, the above holds, but in some respect it does not tell the entire truth, because `glibc` and a new Linux kernel are only required for very broad and general cross compiler usage, e. g. when doing foreign kernel development. For the purpose of writing this thesis and testing "merely" compiler code changes, a simpler and less disk space consuming approach suffices. It is the aim of this section to explain why, and especially how, this more simple solution works.

Binutils. A top prerequisite for building a cross compiler, apart from having the main GCC source code itself, is a package called "GNU Binutils" (see Appendix B). Binutils is a collection of binary tools, including in particular an assembler and a linker to create binary files. Like all the other components involved in building a cross compiler, Binutils have to be configured and compiled *for* the foreign architecture, in the following example for AMD's `x86_64`, supporting Linux:

```
$ ./configure --target=x86_64-unknown-linux   \
              --disable-nls
$ make
```

The current host usually does not need any further specification, although it is possible to explicitly set it by using `--host=i686-pc-linux-gnu`, for instance. `--disable-nls` disables native language support to keep the configuration as straightforward as possible and also to avoid unnecessary interprogram or interlibrary dependencies.

[6] The document is freely available from `http://www.baldric.uwo.ca/HOWTO/`.

GCC. Typically, GCC has to be built in at least two separate steps. The first build is supposed to create a basic C cross compiler which is then used to compile a foreign system library. In the next step, the library gets used to build all the "rest" of GCC. That is, step one depends on the working installation of the foreign Binutils and step two depends (at least) on a successful build of the system library.

Not surprising, there is no bullet proof recipe on how to properly configure the GCC cross compiling build process. However, the following command line options will usually look similar on a lot of different targets:

```
$ ./configure --target=x86_64-unknown-linux   \
              --disable-nls                   \
              --enable-threads=single         \
              --disable-shared                \
              --without-headers               \
              --with-newlib                   \
              --enable-languages=c
$ make
```

In this case, the command will set up a C compiler for `x86_64-unknown-linux`, without native language modules, without support for multiple threads, without support for building dynamically linked binaries, i. e. without references to a system library, and without the presence of operating system header files, but with support for "Newlib" (see also next paragraph and Appendix B). However, `--with-newlib` does not necessarily mean that Newlib is really used; it just modifies the build process so that GCC builds as little of itself as possible, because Newlib contains several functions that GCC normally provides itself.

Unfortunately, most of this is not obvious at all, and to make things worse, the configuration does not work on all targets alike. However, in order to test ones code changes to *just* the C compiler, it is, in many cases, sufficient. This is mainly due to the fact that error messages which may be spewed during the build process are generally not affecting the C back end. So, by the time an error about unsatisfied library dependencies occurs, the main C compiler is already built and ready to generate "foreign assembler", although it can not yet create useful executable binary files.

C Library. Naturally, the cross compiler will depend on various system calls to translate a source file into executable machine code. There are two common ways to provide these system calls to the cross compiler: one can set up and compile GNU's entire C library (see Appendix B), or use a smaller substitute called "Newlib" (see Appendix B). The GNU C library, however, is much larger than Newlib and requires more resources and hence, time to build. Therefore, the choice for Newlib is an easy one, despite the introductory comments in the *PARISC HOWTO*.[7]

[7]It may even be possible to build GCC as a cross compiler with size optimised C system libraries like diet libc (see Appendix B), but that has not been tried by the author. Maybe this could be covered in the future by a proper *Cross Compilation HOWTO*?

Another aspect to consider when setting up a a cross compiler is that a foreign C library is not always required. Firstly, it depends on how much the cross compiler is supposed to do and secondly, not all target systems depend on the library to exactly the same extent. Indeed, for most test cases which occurred during this work, it was not necessary to create a C library at all, in other situations, e. g. when trying to run GCC's extensive test suite on a cross compiler, it was absolutely essential to have a *proper* installation at hand, which is not always trivial to achieve. To cite the *PARISC HOWTO* once more, it states that "Sometimes glibc builds... sometimes it doesn't. Sometimes it's just magic." Indeed, the comment is not very helpful, but it sums up the circumstances quite well.

Nevertheless, the following commands usually work, when configuring a C system library for a non-native platform:

```
$ ./configure --target=x86_64-unknown-linux            \
              --with-headers=$HOME/src/linux/include   \
              --disable-nls
$ make
```

Obviously, this requires access to the kernel header files which, in this example, are stored in the directory `$HOME/src/linux/include`. Newlib, on the other hand, does not necessarily depend on the header files and thus gives yet another good reason to use it. (Building seems faster and more predictable as well.)

Summary. As already pointed out in the beginning of this section, the author was required to set up several different cross compilers for all kinds of targets during this work, e. g. for `arm`, `x86_64`, and `sparc`, amongst others. Each of them revealed their own peculiarities, but in most cases it was, at least, not necessary to set up an error prone system library. Some targets compiled "out of the box" like `arm`, for instance, while others would cause great frustration during very odd stages in the overall build process. But given the huge number of supported platforms, GCC's cross compiling facilities remain impressive. They enable developers to make use of code for a platform which they have no physical access to and even more, they allow stringent tests and bootstrapping. Since this thesis affects not only Intel `ix86`, it would have been barely possible to complete it without GCC's ability to act as a cross compiler. Given the huge number of different possibilities to approach cross compiling, this process can become very time consuming and tedious though, with the subtle contradictions in different online manuals not making the task any easier. Hence, this part of the thesis should not only be regarded as a description of the process, but also as a critical review that lists the hurdles involved. Certainly, there is still room for improvements in this area of GCC.

CHAPTER THREE

EXISTING TAIL CALL SUPPORT IN GNU C

Tail call support has been a part of the GCC back end for several years now, therefore this chapter mainly focuses on its original implementation and the limitations. Because some of those are rather stringent, the GCC community has agreed on their own terminology regarding the subject. Hence, it is also part of this chapter to introduce their terms and explain in detail why projects like the Glasgow Haskell Compiler, currently, do not benefit from the existing form of tail call optimisation of GNU C.

3.1 TAIL RECURSION AND SIBLING CALLS

The preferred way of the GNU C compiler to deal with tail calls is in its back end during basic block analysis (see Appendix A). That is, the back end first determines the *exit block* of a function, and then checks whether that is the immediate successor of the block that the function call is in. In other words, it verifies whether a call in the tail position is *really* the last instruction of a function. However, upon detection, it is not decided whether the call should *really* be optimised, because it needs to qualify as *sibling call* first. (The GCC source code also often uses the abbreviation "sibcall" referring to the same concept.)

Definition (Sibling Call). *Let f and b be two not necessarily distinct functions. A call from function f to function b is a sibling call iff*

1. *it qualifies as proper tail call,*
2. *the number of arguments to b (or the space occupied by them) do not exceed the number of arguments to f (or the space occupied by them),*
3. *the return types of function f and function b match.*

Obviously, this definition covers proper tail recursion as well, because if in a call, f and b are really the same function, then the number of arguments and the return type are guaranteed to be always the same. In all other cases, however, the definition means that the caller's and the callee's function signature have to be at least somewhat similar in order to be regarded as "siblings".

Example 1. The call to function `bar` is a sibling call, because it is in the tail position. The caller and callee accept exactly the same number of arguments and share the same return type:

```
int bar (int, int);

int foo (int a, int b)
{
  ...
  return bar (b - 2, a - 5);
}
```

Example 2. The following code snippet is not a sibling call, because `bar`'s arguments demand more stack space than `foo`'s — in fact, twice as much. The caller and callee are *not* siblings:

```
int bar (int, int);

int foo (int a)
{
  ...
  return bar (a, abs (a));
}
```

Example 3. At a first glance, the function signatures do not seem to match; nonetheless is the argument of `foo` on almost all platforms greater in size, than both of `bar`'s together. Therefore, (on these platforms) the call can be optimised into a sibling call:

```
int bar (int, int);

int foo (long long a)
{
  ...
  return bar ((int) a * 2, (int) a * 3);
}
```

The above examples demonstrate the essence of the definition, but also show how it is partially dependent on the operating system's memory layout, hence ABI. Not all platforms share the same internal representation of C types, so recognising sibling calls, in practice, goes beyond *any* high level definition; also, because operating systems and hardware are constantly subject to change.

It is probably best to consider a sibling call as a "technical term" only, rather than a widely accepted scientific notion. Indeed, recent traffic on the GCC mailing lists has shown that even some of the developers aim to drop their custom (and sometimes very confusing) terminology as sibling calls slowly become more powerful.

3.2 CURRENT IMPLEMENTATION

It should be clear that detecting sibling calls and tail recursion is not only a matter of dissecting a program into basic blocks. If given a sufficiently high level of optimisation, GCC is promoting the information about possible sibling call candidates throughout various different stages of compilation, beginning at RTL generation in the file `calls.c`.

The front end. Currently, the front end does not explicitly know about sibling calls, even though it is theoretically possible to detect and mark at least some of them during this stage. When GCC's parser (`c-parse.in`) detects a function call, no matter of which kind, it always passes it on to `build_function_call` inside `c-typeck.c` in order to start building the corresponding syntax tree and later also RTL code.

The `calls.c` file. For the purpose of this thesis, the file's most important function is `expand_call`. It is responsible for generating all the code for a function call which is stored in a tree expression, and it returns an RTX for its value. Most importantly though, the function already makes an estimate over which calls may be subject to sibcall optimisation later on and, thus, generates additional code for such candidates. To be more precise, `expand_call` fills a pattern called "`CALL_PLACEHOLDER`" with multiple versions of RTL instructions to represent the current function call (see Fig. 1).

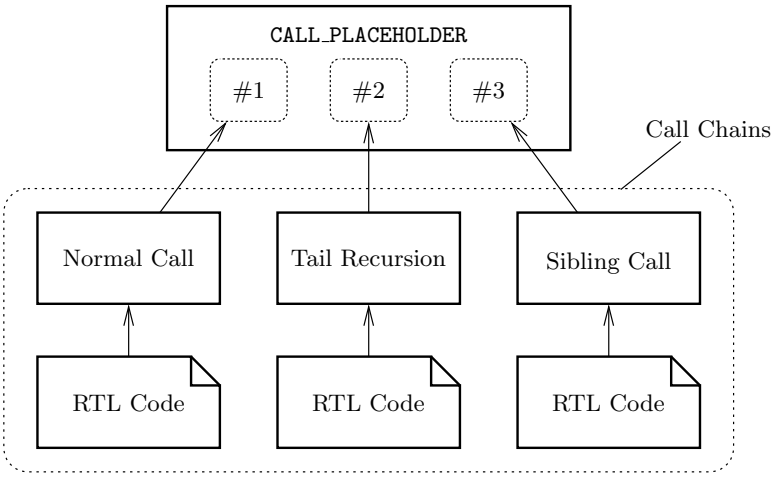


Fig. 1. A `CALL_PLACEHOLDER` pattern contains three different call chains each representing one and the same function call: one chain consists of code for a "normal" call (#1), one is reserved for tail recursion (#2), and another one for sibling calls which, generally, are not tail recursive (#3).

It does so by entering a loop for each call in the tail position, where every iteration creates an individual chain. (For tail recursion there are even multiple ways of optimisation: usually, the compiler will try to convert the tail recursive call into a `goto` statement inside `stmt.c`. Should that fail, GCC will still attempt to convert it into an "ordinary" sibling call, before it finally gives up.)

The `sibcall.c` file. The code in `sibcall.c` is being executed at the end of RTL generation; the compiler selects one of the three call chains, mainly according to the outcome of the basic block analysis. That is, either the call is in the last basic block of the current function, or it is in a block whose successor is an explicit exit block. Due to the inner structure of GCC's basic blocks, the analysis is relatively straightforward (see Fig. 2): a loop runs through the entangled blocks until, either the last one contains a pointer to the RTL code representing the function call (end rtx), or the exit block is reached.

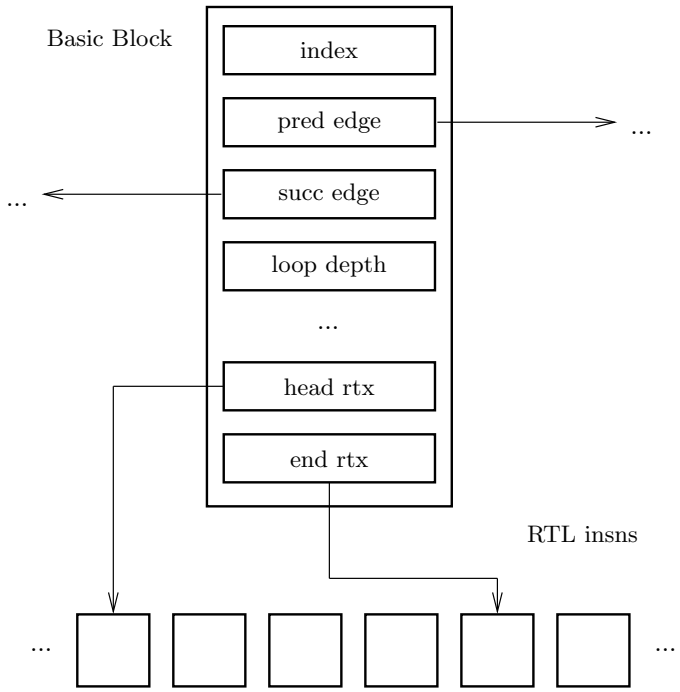


Fig. 2. A basic block consists of various fields and references pointing to other blocks and also to RTL instructions. The picture follows the definition to be found in `basic-block.h`.

Machine-specific code generation. After the compiler has chosen its preferred call chain, it is up to the machine-dependent part to handle any sibling calls by emitting the according prologue and epilogue instructions which, in the end, cause the memory and performance benefits sought (not only) by the functional language community. Those targets supporting the notion of proper tail calls, usually translate the assembly "`call`" into "`jump`" (or to whatever name the according architecture deploys) which does not open a new stack frame for the callee. Of course, if a tail recursive call was transformed into C's `goto` statement earlier on, then the work is already done for this particular recursive call; no further optimisation by the back end is required.

The emitted epilogue differs from the "normal" one as it is defined in the ABI, in the sense that the caller does not need to worry about memory clean ups, once the callee has finished execution. Hence, the alternative epilogue for sibling calls typically contains the following steps:

- The caller stores the callee's arguments in the area for its own incoming arguments. (By definition, this is not part of the epilogue, but without this step the following instructions would not make sense.)
- The caller has to pop callee saved registers, should it be required.
- If not given the option `-fomit-frame-pointer`, the caller restores the frame and base pointer, by reloading both with their prior values. (On Intel `ix86` architectures this can be achieved by issuing only a single instruction: `leave` [Brey 1995, § 6-4].)
- Instead of finishing with a generic function call, the callee is invoked via the architecture's low level `jump` command to reuse the existing stack frame along with the stack slots reserved for incoming arguments.

How sibling calls work in practice. The prologue does not need to be modified, because the callee does not know whether it has been called as an ordinary function, or whether it was invoked via a jump command. It may still reserve space for locals, find its arguments in the expected area on the stack, saves callee saved registers, and so on. When it has finished execution, however, it does not return control back to the caller; instead, the program continues with the function that has originally invoked the caller, as can be seen in Fig. 3.

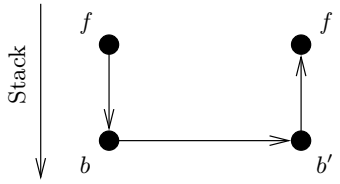
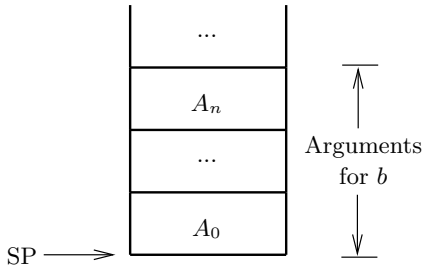


Fig. 3. Function f issues a normal call to b which in turn "sibcalls" function b'. When b' has finished execution, it frees some stack memory and returns to f.

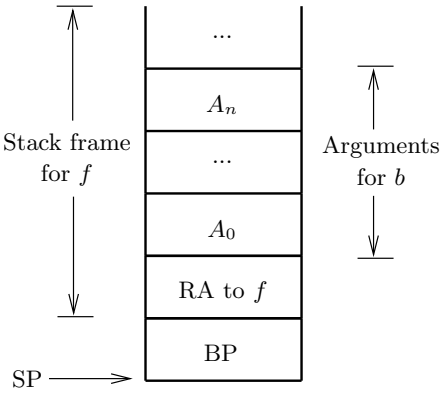
This is because the return address put onto the stack by function f for function b does not get touched until the end of the *whole* call sequence, where it then gets removed. So to function b' it looks like it has been invoked by f, not b. Hence, in the end it returns to f.

In order to understand what is really going on when a sibling call is emitted, one has to take a closer look at the run time stack. All the following examples depict the situation on an Intel `ix86` architecture, unless explicitly marked otherwise. Therefore, each stack slot is 4 bytes in size. SP labels the stack pointer (`%esp` on 32-bit Intel platforms), BP represents the base pointer (`%ebp` on 32-bit Intel platforms; also known as the frame pointer), RA denotes the return address, and A_n stands for the n-th function argument.

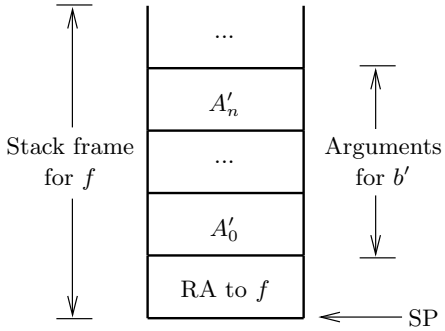
Figure 4 shows how the before mentioned calling sequence (see Fig. 3) does *not* build up stack space by first opening a frame for f, then for b and finally for b'. The nature of sibling calls is reflected in the reuse of the incoming argument and current stack space of a function. Unlike in "normal" calls, the only stack frame present on the stack during the entire calling sequence, is f's.



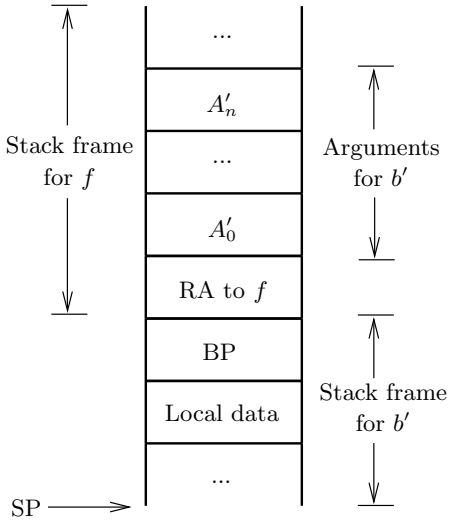
(a) The above figure shows the bottom of the run time stack as it looks when f is about to issue the call to b. All of b's arguments are part of f's stack frame.



(b) After issuing the call to b, b stores the base pointer below the return address and begins computation. Here, BP is the *only* occupied stack slot of b's stack frame, so far.



(c) Almost at the end of execution, before b issues the sibling call to b' it shrinks the stack by restoring the base pointer and reuses its very own incoming argument space for b'. Notice, how the return address to f remains unchanged by this step.



(d) Now b' can continue as usual by pushing the base pointer, reserving space for local variables, and so on. Finally, when it returns, the only choice is f (RA to f), because in a sense, b has already "returned" by issuing the sibling call to b'.

Fig. 4.

Saving one stack frame may not seem like much of an improvement, but if this example is stretched out to mutual tail recursion, for instance, then the gain can make a remarkable difference. In fact, if the depth of recursion is sufficiently high, such a (sibling) call mechanism would be GCC's only solution to prevent a system's run time stack from overflowing.

3.3 LIMITATIONS

The criteria for successful sibling call optimisation on the RTL level are quite strict though and, at this stage, these do not even take further constraints from the underlying hardware into account; those are considered in a separate step, afterwards. It was one major objective of this work to loosen these criteria. Hence, the following elaboration will describe the situation as it was *before* the author's results were officially adopted by the maintainers of GCC.

In `calls.c`, the routine `expand_call` contains most of the (partially very complex) `if` conditionals, making sure that only the "right" optimisation candidates are chosen. The following paragraphs explain each condition one at a time and outline the rationale behind it. In summary, those are:

a) No pending clean ups

b) Matching argument sizes of caller and callee

c) No variable argument functions

d) Sibling call epilogue must be defined

e) No struct returns

f) Caller and callee must have matching return types

g) No `setjmp` and `longjmp`

h) No volatile functions

i) The caller's stack frame must be empty

j) No indirect calls

k) No position independent code

l) No overlapping arguments

m) (Partly) Machine-dependent constraints

a) No pending clean ups. Foremost, this constraint affects C++ users, because a call in the tail position may not be the last instruction if it is followed by class destructors, often containing code to free memory, for instance. So any pending clean ups will, of course, prevent sibling call optimisation. A programmer, however, will not find this constraint very predictable at all times.

b) Argument sizes must match. More accurately, the caller's arguments must at least be as great in size as the callee's.[1] From all present constraints, this is the most difficult one to overcome, because in order to be able to issue a sibling call to a function with larger argument space requirements, the caller would have to shift (at least) one stack slot containing the return address. While additional move statements are not a huge problem, their implications are, because they would require the introduction of an outright new calling convention (see also § 5).

The call sequence illustrated in Fig. 3 clearly demonstrates the problem, if the following function signatures are assumed:

```
int f  (int, int);
int b  (int, int, int);
int b' (int, int, int, int);
```

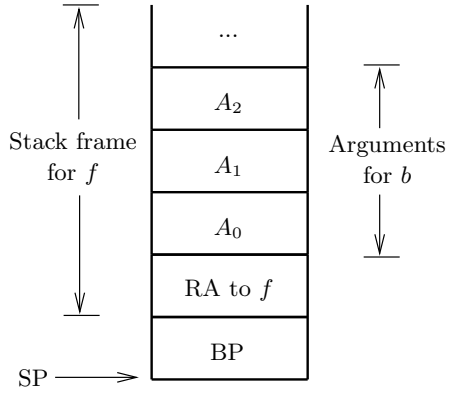
To have the example no more complicated than necessary, the functions only handle integer values and it is assumed that the compiler accepts a name like `b'` which is not common C, of course.

Figure 5 shows how a sibling call from a caller with less arguments than the callee would affect the consistency of the run time stack. However, such a call would not be possible with GCC, because this configuration constitutes a serious problem in terms of memory deallocation: when b' returns, function f would only attempt to clean up three arguments instead of four, because it has no information about b'. To f it appears as if only function b and not b' has been called. On the other hand, if GCC would allow b' to clean up its own argument space to solve f's problem, it would loose compatibility with the C Calling Convention which clearly assigns the responsibility for the callee's arguments to the calling function.

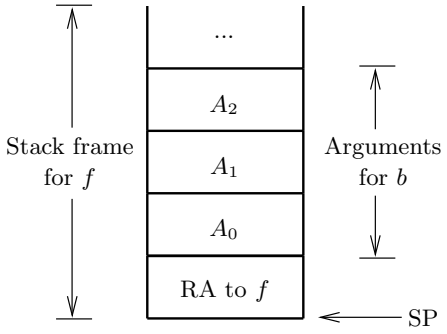
Not even the `stdcall` convention has the potential to improve this situation, because even if b' cleaned up its own argument space, f would still need to know about it in advance, so that it does not try to wrongly unwind the stack. If the call chain is sufficiently complex and long, it would imply that a lot of program functions have to use the alternative calling convention, because every function of the sequence would be required to clean up its own argument space. As a matter of fact, GCC is hardly able to make any predictions about which functions end up in such a call sequence and which do not. The important point is that, often the entire program would need to deploy the alternative convention.

This scenario may sound odd, but it is possible. In fact, it is even possible to use a calling convention along the lines of `stdcall` to potentially enable a greater range of sibling or tail calls, but breaking with the standard C Calling Convention also means giving up compatibility with any other C program for external linkage, save those compiled with the alternative convention in place. However, an unmodified `stdcall` is not suited for this task, because its epilogue instructions do not perform the essential shift operations on the stack. (Return address, registers, etc.)

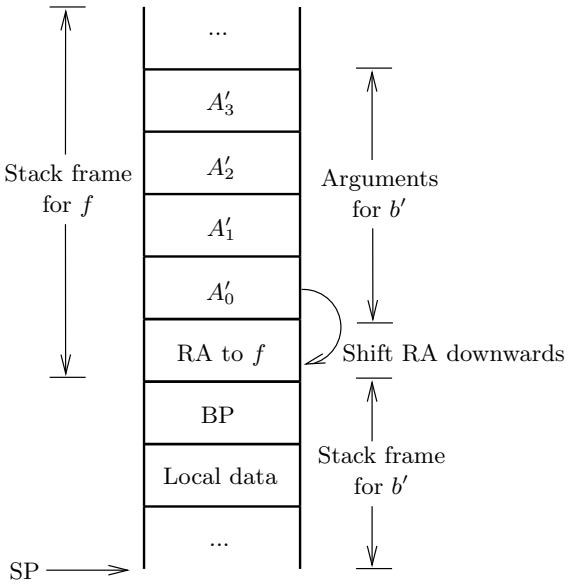
[1] The original requirement was even more stringent: siblings were only those functions whose signature was *exactly* the same, thus the unusual, but once appropriate, terminology.



(a) This figure depicts the run time stack, before b is issuing a sibling call to b'. In this situation, the stack frame reserved for function b consists only of a single slot, the base pointer.



(b) Normally, using sibling calls, b would first clean up its own stack frame (i. e. pop the base pointer) and have the arguments for b' ready in the own incoming argument space. There is, however, not enough space for four arguments.



(c) In order to fit all of the arguments for b', it would be necessary to shift the return address (RA to f) down. That is the only way to pass all of the arguments via the stack.

Fig. 5.

c) No variable argument functions. It is not possible to optimise calls to functions taking a variable amount of arguments, such as `printf` for example. In order to explain why, one has to examine yet another example which is given in Fig. 6. Let b' be a function that takes a variable number of arguments. If

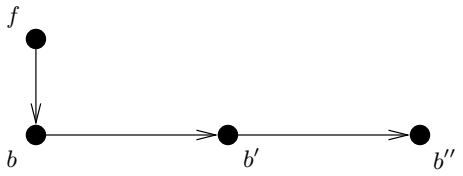


Fig. 6. A call sequence of one normal and two sibling calls.

$b \longrightarrow b'$ is a sibling call, then the next potential sibling call $b' \longrightarrow b''$ would be impossible to realise even if b'' accepts a fixed number of arguments, because b' has no information about how many arguments it has been invoked with. As a consequence, this constellation only works if b, b', and b'' all expect a fixed number of arguments.

d) Sibling call epilogue must be defined. A minimum requirement for every target is the definition of an according sibcall epilogue as part of the machine description. Although, most of the optimisation is performed on the RTL level, a target must have access to alternative epilogue instructions in order to emit corresponding machine code. Currently, not all GCC supported systems offer sibling calls, and those who do put further, greatly varying constraints on their functionality. But very often, the constraint has more to do with the number of people supporting a platform, rather than any physical shortcomings of the underlying hardware or operating system.

e) No `struct` returns. Functions returning large C structures, will not fit an instance into the usual return register `%eax`. Since C is always call-by-value, the ABI requires the caller to reserve space in its own stack frame for the return value of the called function. The address of that area is given as a "hidden" first argument and is also given back by the callee in register `%eax` for future access [The Santa Cruz Operation 1996]. Interestingly enough, the callee removes the "hidden" argument from the stack, unlike all its other parameters.

The reason why structure returns break sibling call optimisation is more of a pragmatic nature, rather than a question of whether it is possible or not. This is based on the gross assumption that, in reality, only a few tail calls would be made to functions returning a `struct`, as a comment in `expand_call` explains:

```
/* Doing sibling call optimization needs some work, since
   structure_value_addr can be allocated on the stack.
   It does not seem worth the effort since few optimizable
   sibling calls will return a structure.  */
```

Again, the situation is similar to the one described in **b)**. If a caller emits a sibling call to a callee returning a struct, it has to shift things around the stack,

because it can not allocate space for the structure in the "usual" location as that is going to be reused. Hence, it would have to add space to the *previous* stack frame and shift down every live stack slot in between. While this is possible, it would not be desirable *if* there is hardly any code that would benefit from it. In fact, the issue could become a lot more complicated when taking further ABIs into account as some may use a call clobbered register to store the structure's location and those can not be guaranteed to be usable on all platforms alike, especially when the tail call is indirect (see **j)** for further details about call clobbered registers).

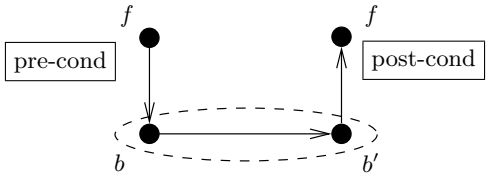
f) Same return type. Basically, this requirement is a consequence of **e)** and, at the same time, very similar to **b)**; it translates to the caller and the callee having to share the same structural type equivalence, hence return mechanism. For example, the ABI for Intel `ix86` architectures predefines the return register `%eax` for 32-bit values and requires C types like `long long` which take up 64-bit to be split into `%eax` for the low part and `%edx` for the high part [The Santa Cruz Operation 1996]. Therefore, sibling call optimisation is possible as long as both functions agree on the same mechanism or the same registers for returning their values, because (on a lot of popular platforms) the compiler makes no difference as to whether there is a character or an integer in the return register; both fit equally well.

Figure 7 helps to illustrate this situation: the main point is that, to the top most calling function f, it has to look like the tail call "has never happened", i. e. the pre-conditions upon calling function b must correspond to the post-condition when the sibcall sequence returns control back to f[2].

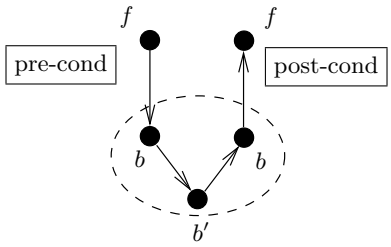
g) No `setjmp` and `longjmp`. In UNIX, `setjmp` and `longjmp` are the commands to handle signals like `Ctrl-C` interrupting a program in execution. They are part of the standard system library and provide a way to avoid the normal function call and return sequence, typically to enable an intermediate return from a deeply nested function call [Kernighan and Ritchie 1988, § B8]. When `setjmp` saves state information like program and stack counters, general purpose registers, etc. it uses an instance of the `jmp_buf` data structure for `longjmp` to restore, after a signal has been processed. In effect, `longjmp` allows one to jump over the calling stack to any previous frame beginning at the next instruction past the originally called `setjmp`. Even though that sounds ominous in itself, due to the fact that sibling calls delete stack frames for other functions to reuse, the main problem is that `longjmp` typically means a function can return more than once.

GCC does not add such functions to the `CALL_PLACEHOLDER` pattern — not only to simplify sibling call optimisation. Instead, instructions for a normal call are generated. From a technical point of view, however, it is possible to

[2]To understand the prerequisite, it may help to compare it roughly with the "while rule" in the Hoare Calculus [Winskel 1993], better known for the concept of the *invariant* which has been introduced by this rule. The invariant is a condition which holds upon loop entry, and after it has finished computation.



(a) A typical sibcall sequence: function f calls b which sibcalls b' and returns to function f.



(b) Here, the same function f calls b which calls b' and returns normally. Unlike in the first figure, there are no sibling calls involved in the depicted call sequence.

Fig. 7. Two different call chains, involving the same functions: the dashed area is the interesting part of the sequence, because function f is not supposed to "notice" what happened in that area; it called b under certain pre-conditions (expectation of return types, required argument space, etc.) and will end up in a predestined post-condition, i. e. "pre-cond" of situation (a) must match "pre-cond" of situation (b); the same holds for "post-cond".

use `longjmp` and sibling calls, but the work required may not seem worth it, especially as long as volatile functions are not supported which impose a very similar problem (The next paragraph elaborates more on this subject.)

h) No volatile functions. Still, to many peoples' surprise, the keyword `volatile` is standard C and can be considered as the opposite of the better known keyword `const` [Kernighan and Ritchie 1988, § A8]. Functions declared as being volatile may not be sibling call optimised. The rationale behind that does not originate from peculiar ABIs or hardware design issues, but from a "misinterpretation" of such functions in GCC itself. When GCC encounters a volatile function, it assumes this function does not return. Examples are `exit` or `abort`, but also any other user defined function which is marked by the according attribute:

```
void foo () __attribute__ ((noreturn));

void foo ()
{
  ...
  exit (1);
}
```

GCC versions prior to 2.5 required a slightly different declaration of volatile functions:

```
typedef void voidfn ();
volatile voidfn foo;
```

Both pieces of code represent the same situation, but use a different notation; and they share also the same problem: within the basic block analysis, sibling calls to `noreturn` functions do not have an explicit edge to the function's or program's exit point, which means that the code intended to insert the alternative sibling call epilogue would not get executed. Consequently, GCC prevents sibling calls for such cases which is not a terribly huge burden for a programmer or a front end, because most functions, indeed, do return. In the future, it may be possible that this situation will be improved, but at the moment the pressure for doing so, does not seem high enough.[3]

i) Empty stack frame. The original sibling call optimisation always had the potential of handling functions whose stack frame was *not* empty e. g. occupied by the declaration of local variables or temporaries. However, GCC currently prevents sibling calls for these cases, with the result that sibling calls can be used to efficiently compute the Fibonacci numbers, but not for too much more.

The reason for this has already been given in § 1.5: in C, it is possible to have global references to a local variable. If the calling function uses such a reference as an argument, the stack frame must still be live for the callee's access to that parameter. Sibling and proper tail calls, however, are supposed to delete the current stack frame for reusage. Thus, GCC would have to detect such cross references, should it allow automatic sibling call optimisation for functions with local variables.

Clearly, this is one of the most limiting constraints so far. It means that a vast number of real world C applications have the possibility to issue sibling calls and save stack space but, in reality, almost never get a chance to do so. This is mainly because a C function without arguments is usually rather limited in its possibilities and often serves merely as a predicate returning "true" or "false" for a certain input condition. (It can be argued whether it really matters if these smaller functions are optimised at all, or not.) The big, heavyweight — also in terms of stack space consumption — functions, however, are not even considered as candidates, even though they constitute the most crucial cases for the optimisation.

At least the GCC maintainers are well aware of this weakness in the current implementation, though, until now, no one has volunteered to pick up work on a sophisticated liveness and flow chart analysis that would solve the problem. But even without such a complicated detection in the back end it is possible to address this issue. In § 4 a possible solution is sketched which, in this aspect, is largely based upon extensions in GCC's front end.

j) No indirect calls. In order to support continuation passing in GHC, the back end is required to offer optimisation for indirect function calls, since they

[3] The notorious exception to this rule of thumb is the Mercury Compiler (see Appendix B) which declares many functions as volatile to generate better and faster code with GCC serving as its preferred back end [Conway et al. 1995, Henderson et al. 1995].

constitute the program's continuation. Sibling calls were designed, with only direct calls in mind though, and if the "Evil Mangler" (see § 1.3) would not have been invented, almost none of the GHC generated C functions could be translated into a sibling or properly tail recursive call. In other words, GCC practically provides almost no sibling call support for GHC at all, even though it serves as its most portable back end.

Certainly, it is no exaggeration to say that, this is a suboptimal situation: a widespread compiler's portability depends strongly on a Perl script to prevent the run time stack of its compiled programs from overflowing. However, understanding the rationale of this constraint also means understanding the constraints of the underlying hardware and the corresponding ABIs.

Depending on the number of arguments and used registers, assembly code for a direct sibling call from `foo` to `bar` (e. g. on 32-bit Intel platforms) looks typically like this:

```
foo:
        ...
        movl    %eax, 8(%ebp)       Prepare a function argument.
        popl    %ebp                Pop base pointer.
        jmp     bar                 Jump to function bar.
```

Here, the jump command's one and only parameter is the target function name, `bar`. An indirect call, however, redirects to an address which points to the first instruction of the callee. So instead of jumping to a label, `foo` would have to jump to an address, held in one of the machine's free registers:

```
foo:
        ...
        movl    %eax, 8(%ebp)       Prepare a function argument.
        movl    ptr, %ecx           Move destination ptr into %ecx.
        popl    %ebp                Pop base pointer.
        jmp     *%ecx               Jump to the address stored in %ecx.
```

Even though this is valid assembly code, it has, until now, only been wishful thinking, for GCC would have never produced code like this. In such a case, it would rather use the ordinary `call` command which opens a new stack frame for the callee.

One of the architectures which required such a restriction is ARM. On ARM, which is an ever popular RISC platform, since companies like Apple, Palm, and Compaq based their handheld devices upon it, all call clobbered registers are used for argument passing [ARM 2000]. It is the nature of sibling calls that the calling function restores all callee saved registers to the state its caller expects, *before* calling the target. All argument registers, on the other hand, have to be loaded as the called function expects, so in calls with at least four arguments there is no space to hold the address. Plus, in many cases additional scratch registers are needed for marshaling the parameters and values.

These problems do not occur in direct calls, even on ARM, because the address can be conveniently stored in any call clobbered register. If its value

was important to a calling function, it would be temporarily stored on the stack and restored when the whole call sequence has come to an end.

The popular Intel 32-bit platform, however, is one of those which does not depend upon free registers as much when calling a function; registers are not all assigned a certain role during the calling procedure. Moreover, there are various different registers for the caller to chose from when storing an address, e. g. `%ecx`, `%edx`, or even `%eax`. Either they are all available at once, or the back end can determine at least a single one to store the required address.

For GCC, it means that handling of indirect sibling calls is impossible in the RTL generation stage where most of the optimisation decisions are made. During this phase, it can not determine or even react properly to peculiarities of the according ABIs. Hence, the indirect sibling calls are always disabled by default, regardless of the host or target platform which may have the capabilities to support this important concept.

k) No position independent code. Some platforms allow programs to link dynamically against libraries in order to save main memory when various different applications all depend on the same functionality. In such cases, the library is loaded once and mapped into each of the program's virtual memory as if it was the only application on a system. On UNIX and also on some other systems, this is achieved by using position independent code for libraries, so that addressing is *relative* to an offset, rather than *absolute* to a fixed address space.

On many platforms, however, position independent code must not be sibling call optimised. The problem involved is that the offsets for navigating through the program have to be kept in registers; on Intel 32-bit, `%ebx` is used which is a callee saved register; other platforms follow that notion. Unlike functions using normal calls, those deploying sibling calls have to restore the callee saved registers *before* branching off to the subroutine, not when they return themselves. Normally, that is no problem, because at this point, the top most calling function does not care for the value stored in `%ebx`, but the position independent code depends on this value upon each and every jump, call or branch command. In other words, restoring `%ebx` in a sibcall epilogue would basically deliver the wrong offset and the program would abort.

l) No overlapping arguments. On a closer look, the essence of the current sibling call implementation is the reuse of the incoming argument space, rather than the current stack frame, as it is usually greater in size. Also, according to **i)**, the stack frame offset must be *zero* to issue a sibling call.

Unfortunately though, having a nearly empty stack frame also means not being allowed to have space for temporaries and additional slots needed to marshal arguments for the callee. Hence, if the outgoing arguments depend solely on the values of the incoming arguments, they can not be overwritten, because that would delete the previous value. That is, in many cases which, to make things worse, are often not obvious to a programmer, a sibling call is prevented because the caller's and the callee's arguments overlap and possess

a cyclic dependency. Instead, the arguments have to be stored on the run time stack violating the empty-stack-frame rule.

m) Other possible constraints. There are further reasons for sibling call optimisation to fail, aside from the mostly platform independent issues. When an optimisation candidate is considered by the underlying machine and operating system dependent code, there are various obstacles upon which GCC switches back to the "normal call chain", instead of using the more efficient sibling call. Sometimes, it is difficult to differentiate clearly between the platform dependent and general restrictions as some of them are closely tied together. Subtle differences do exist, and they require thorough checking in the machine dependent part of the back end.

To give one example for the Intel 32-bit architectures, sibling calls are not possible for functions which return a float value on the register stack of the Floating Point Unit 80387, if the calling function is not doing the same. Similar to the requirement of matching return types, such two functions would expect different stack adjustments once they return.

The list could be continued, but the presented restrictions should be sufficient to demonstrate the involved problems and to show where most of the work and time is needed so that the situation can be improved.

CHAPTER FOUR

POSSIBLE ENHANCEMENTS

> Normal people... believe that if it ain't broke, don't fix it. Engineers believe that if it ain't broke, it doesn't have enough features yet.
>
> — Scott Adams, *The Dilbert Principle* (1996)

During this work a lot of time was spent elaborating on different solutions and scenarios to improve tail call support in GCC. Given the relatively huge set of constraints and the complexity of the current sibling call implementation, it is not obvious which approaches would lead to good results and which would introduce new, unforeseen draw backs.

The purpose of this chapter is to outline several ideas that were examined during this work and to discuss their implications in terms of portability, maintenance, and related issues. A solution labelled "Super Sibcalls" (see § 4.2) is even partially implemented and the according source code changes included with this thesis (see also Appendix C).

4.1 DESIGN ASPECTS

In § 1, the most important requirements GHC imposes as a front end for a functional language, were already outlined, e. g. indirect sibling calls to support a form of continuation passing. Therefore, the following elaboration is more concerned with requisites of the back end itself, so that improved facilitation of tail calls in GCC is possible at all.

Foremost, the presented ideas are a direct result of the constraints examined in detail in § 3.3. However, not all of these constraints can be addressed with equal effort. That is, improvements should be made to those restrictions that do not directly depend on each other and that do not significantly affect portability, otherwise it would be difficult to justify the changes to the maintainers of GCC.

Hence, the important question is "In *what way* can (and should) the back end be enhanced for sibling calls to become more useful and general?" Various different approaches and scenarios are possible to answer this question, as the following examples show:

a) Ignore machine-dependent aspects. Obviously, not all platforms are equally as important when it comes to today's software development reality.

The number of people still actively using a Motorola 68000 system to write code on, is likely to be smaller than the number of programmers relying on a 32-bit Intel platform. Therefore, it could be argued that changes, which may break the existing sibling call optimisation on a less known architecture are acceptable, if a commonly known platform, at the same time, would experience significant benefits.

Indirect function calls are a perfect example for such thinking. As this thesis will show, their optimisation is quite possible on (say) modern Intel or AMD platforms, but imposes great difficulties on an implementation for ARM, or any other platform deploying a similarly restrictive calling convention. ARMs, however, although not generally found inside desktop computers have widespread use in other environments, e. g. in embedded systems.

In consequence, enabling indirect sibling calls, in general, during the platform independent RTL generation is not a good idea, because this would sacrifice portability; a chief requirement for all changes to RTL processing code is *portability*.

b) Wrapping the modifications. Sibling calls are only useful, if at least basic compatibility to existing programs and libraries is given — somehow. That is, either they fully comply to the C Calling Convention (or whatever convention a system demands), or they introduce another layer of abstraction, making the changes transparent to other processes.

Since proper tail recursion and the C Calling Convention do not mix very well, one way to interface the two would be a *function wrapper*. Intentionally, it would be used for those functions which are called in the tail position. While it would be relatively straightforward to implement such an alternative entry point for a function, it would be rather difficult to emit code for the corresponding exit code which would need to adjust stack frames and pointers, because the callee and caller may have very different signatures. So, the idea may sound tempting but has also significant drawbacks. Some of the more obvious ones are:

- Similar to **a)**, a wrapper would be highly platform-dependent.
- De facto, the introduction and implementation of an alternative "wrapper epilogue" requires no less work than making up a wholly new calling convention from scratch.
- To "clean up" after functions return, the wrapper code would be required to store temporaries on the stack representing parameter information, or return values.

In other words, inventing an alternative function entry and exit code is, in effect, the introduction of a new calling convention though, in scientific terms, not quite complete to be regarded as such. So, the amount of work required to introduce a wrapper around such changes could turn out to be even greater than dealing with *only* the modifications of a previous calling convention. Plus, implications of such code changes on the already existing parts of GCC must be

foreseeable, but at such an early stage of planning it is not quite clear, whether the function wrapper would be ideal to capsule the concepts in a way that they do not interfere with the compiler's portability and flexibility.

In summary, one could say that wrapping a calling convention very much defeats the purpose of deploying this convention.

c) Invent a new calling convention. The technical realisation of the existing sibling call optimisation could lead one to the proverb "Never touch a running system." Therefore, it has to be examined whether general tail calls are easier to implement by introducing a custom calling convention rather than by extending what is already there and works for a limited number of cases. Sometimes, the effort of trying to understand a current mechanism and later modifying it can be higher than creating a new solution from scratch. A more detailed analysis of this idea can be found in § 4.3 though.

d) Consult the front end for help. Despite (or because of) the careful examination of GCC's back end in previous chapters of this work, one might be interested in figuring out ways to shift the work load, required to enhance sibling call detection and optimisation, partially into the front end, e. g. by syntactic analysis, or by introducing new language constructs directly supporting the concept (and, naturally, against all existing standards). While there are cases in which this could be a real advantage, it is often hard to realise, because one needs to have *very good* reasons to add custom keywords on top of the ANSI-C standard, for example. However, GNU C understands another standard as well, the GNU-C standard which supports nested subroutines, for instance. So, despite all doubt, non-ANSI compliant front end modifications are possible, if they are thought through and if pressure from GCC clients and developers is high enough for the maintainers to adopt them.[1]

e) Sacrifice elegance and performance. The detailed examination of all the obstacles preventing sibling calls in GCC's back end could also lead to the conclusion that trade offs have to be made, when trying to remove some of them. For example, optimisation fails in cases where the function arguments overlap, but only because GCC elegantly marshals values without allocating any extra space for temporaries; a very efficient and also portable mechanism, but unfortunately, at the same time, very restrictive. In such cases, a decision has to be made whether the performance losses would be acceptable if e. g. temporaries are allowed and cleaned up *before* issuing a sibling call. Clearly, this would be less elegant and requires a few extra move instructions as well as memory, but it may have the potential to solve the overlapping-argument problem at the expense of run time speed. Such *trade offs* play, in fact, a big role in sibling call optimisation and, naturally, for this work. (§ 3 has shown that, in theory, far better support for sibling calls is possible, but the trade offs

[1] As a matter of fact, Appendix C contains a patch which does not conform to ANSI-C, but lets the front end help with tail call detection. Not surprisingly, it can *only* be found in this work, not in GCC.

made between the back end's portability and functionality issues have lead to a "minimal" sibling call implementation, in favour of portability.)

Summary. More aspects could be taken into account when thinking of a good and sound solution for GCC's "tail call problem". Each programmer (or client) will impose different requirements on a solution though, because some may not care whether it is portable or not, or whether the GCC maintainers consider the approach broad enough, etc. However, the goal of this thesis was to achieve improvements that are permanent, i. e. that find acceptance in the free software community and by the maintainers of GCC alike. Hence, portability, compatibility, elegance and the possibility to realise the necessary code changes within a relatively tight schedule were, indeed, most relevant for this thesis to succeed.

4.2 SUPER SIBCALLS

The idea for *Super Sibcalls* comes as a direct result from analysing the tail call restrictions and drawing a couple of pragmatic conclusions. The motivation was to sacrifice a certain degree of elegance and, probably, even performance to overcome more severe constraints.

4.2.1 Concept

Super Sibcalls are meant to add to the already existing function call mechanism of GCC in a sense that the back end would create *four* instead of just *three* different call chains, one of them being the Super Sibcall. Their main features are a) no overlapping stack arguments, and b) allocation of extra argument space in the caller's stack frame for marshaling the values. Hence, Super Sibcalls are supposed to work even if the stack frame offset is not *zero*, though the declaration of locals imposes the same problems as it does on the ordinary sibling call mechanism.

This can be achieved by combining the existing sibling call epilogue with additional, mostly platform-independent instructions; chronologically, these are:

- Until the very end of RTL code generation for a particular call, the calling function "assumes" an ordinary call, hence reserves stack slots for outgoing arguments.
- The caller stores the arguments which are, from the perspective of the Super Sibcall, in their temporary position.
- In a separate step, all arguments will be copied into the caller's own incoming argument space.
- The caller has to pop callee saved registers, should it be required.
- The caller restores the frame and stack pointers.

- A low level jump command is used to branch off execution to the callee. (Basically, this is also what happens in the current sibling call mechanism.)

The following example call sequence will help to illustrate how Super Sibcalls work in practice. The functions f, b, and b' all accept four 4-byte parameters each (assumed is, like in previous examples, the Intel 32-bit run time stack):

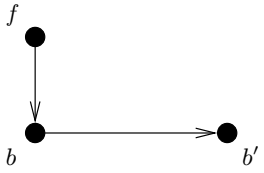


Figure 1 depicts this situation, in particular when b is about to call b': b allocates additional slots for the callee's arguments. The space is required for b to temporarily store outgoing parameters, $T_0 - T_3$. Finally, before the execution branches off to b' all temporaries will be copied back in the incoming argument space of b, $A_0 - A_3$, and the "top most" stack frame can be freed.

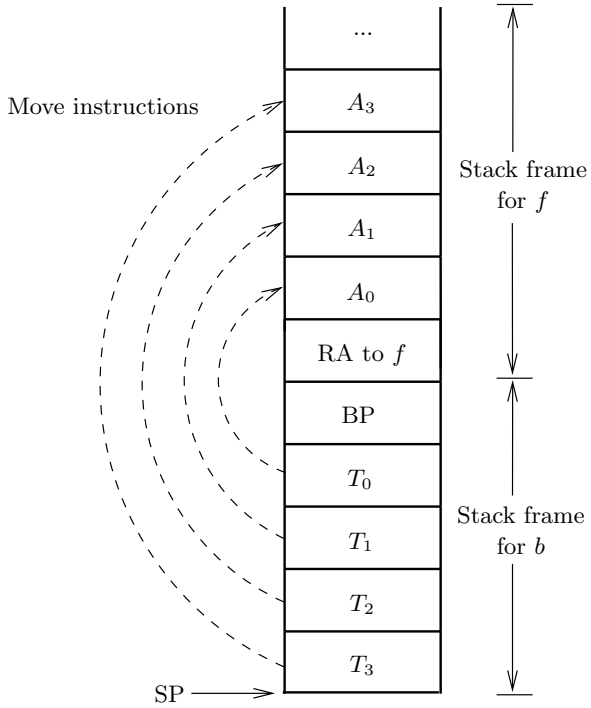


Fig. 1. The Intel `ix86` run time stack and Super Sibcalls; instead of copying the outgoing arguments straight back into the own argument space of b, the parameters are assigned temporary slots where they can reside until the jump occurs.

4.2.2 Implementation

Super Sibcalls were implemented as a prototype only. That means, the modifications were made to prove that the concept works and to see where the

disadvantages are, but especially to discuss a working model with other GCC developers via the mailing lists.

To speed up development, they were based upon a rather crude extension in GCC's parser to explicitly mark calls as being "super". For that purpose a new keyword called "`__tailcall`" was added that should allow front ends and programmers to explicitly annotate a tail call in which no local variables are required to be alive after the call. This is a prediction the back end can not make, presently.

The first, most important step of the implementation was to tag the tree representation of the callee by adding a flag to the `tree` data structure called "`tailcall_flag`", and a corresponding accessor macro, `TREE_TAILCALL`. Next, it is necessary to extend the definition of the `CALL_PLACEHOLDER` pattern in `rtl.def`. GCC uses its own, custom format to describe patterns and instructions, but it is fairly intuitive. Hence, it was merely necessary to add a single "`u`" to the `CALL_PLACEHOLDER`:

```
DEF_RTL_EXPR(CALL_PLACEHOLDER,"call_placeholder","uuuuu",'x')
```

The letter "`x`" states that all instructions contained in this object are RTL expressions (where an "`i`", for example, would represent integers). However, great care had to be taken when handling the modified placeholder pattern, because the optimisation in the back end needs to be prepared, dealing with yet another call chain, otherwise it would be "cut off" in the end. Since this is a rather mechanical process with *a lot* of little "hacks" involved, it will not be explained here.[2]

Basically, Super Sibcalls are being processed (almost) like ordinary calls up to the point where normally the epilogue would be inserted. Then the RTL generating code in `calls.c` determines the final destination for all arguments and simply adds the required move instruction for shifting each parameter into the correct place:

```
for (i = 0; i < num_actuals; i++)
  {
    ...
    /* First, determine offset.  */
    rtx slot_offset = ARGS_SIZE_RTX (argsi.slot_offset);

    /* Now, determine real address.  */
    rtx addr = plus_constant (virtual_incoming_args_rtx,
                              INTVAL(slot_offset));

    /* And finally, compute the correct stack slot.  */
```

[2]Theoretically, it would have been possible to simply replace the sibcall chain with Super Sibcalls, but sibcalls are more elegant and also faster and should still be offered in all those cases where feasible. Super Sibcalls should be regarded as an add-on, not as a replacement.

```
    rtx dest = gen_rtx_MEM (argsi.mode, addr);

    /* Shift argument.  */
    emit_move_insn (dest, src);
  }
```

A problem that has to be addressed when shuffling around arguments is the byte alignment. Each ABI defines how data types are mapped into memory and to what boundaries they need to be aligned to. On 32-bit Intel platforms, data types are either 1-, 2-, or 4-byte aligned (see Table 1), but a compiler is

Table 1. SCALAR TYPES[1]

Type	C	`sizeof`	Alignm.[2]	Intel `i386` Architecture
Integral	`char` `signed char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed halfword
	`unsigned short`	2	2	unsigned halfword

	`int`
	`signed int`
	`long`
	`signed long`
	`enum`
		4	4	signed word

	`unsigned int` `unsigned long`	4	4	signed word
Pointer	*any-type* `*` *any-type* `(*) ()`	4	4	unsigned word
Floating Point	`float`	4	4	single precision (IEEE)
	`double`	8	4	double precision[3] (IEEE)
	`long double`	12	4	extend. precision (IEEE)

[1]Source: *System V Application Binary Interface/Intel386 Architecture Processor Supplement* [The Santa Cruz Operation 1996, § 3]
[2]Alignment is given in bytes.
[3]On the Intel architecture it is not required to have doubleword alignment for double precision values.

allowed to impose a different set up on the run time stack, according to the deployed data structures, or specific features of the hardware. Besides GCC, other compilers use this "freedom" to improve their generated code, as can be concluded from the following quote, taken from the *Intel Technology Journal*, issue Q2 [Wolf III 1999]:

> "The Intel C/C++ Compiler. [...] Data must be 16-byte aligned to obtain the best performance with the Streaming SIMD Extensions of the Pentium III processor. In addition, exceptions can occur if the aligned data movement instructions are used, but data are not properly aligned."

For example, under certain circumstances, a single integer value can be spread across 16 bytes of stack space: 4 bytes to represent the actual value and 12 bytes for padding [The Santa Cruz Operation 1996]. Commonly, this happens when a function declares only a single local variable, because for the Intel `ix86` architecture, GCC distributes data types usually into 16-byte chunks. (Specific optimisation switches, however, can be used to alter such alignment constraints.) Apparently, GCC uses the following formula to calculate a function's frame offset O, including padding.:

$$O = L + P_1 + A + P_2$$

(On Intel) L is the space required by the data types of local variables; P_1 is the padding required to align them to 16-byte chunks; A is the space taken up by arguments and P_2 is the padding around them. This formula is important, because Super Sibcalls must know which parts of the stack frame are arguments, saved registers, padding, etc. An example is given in Fig. 2.

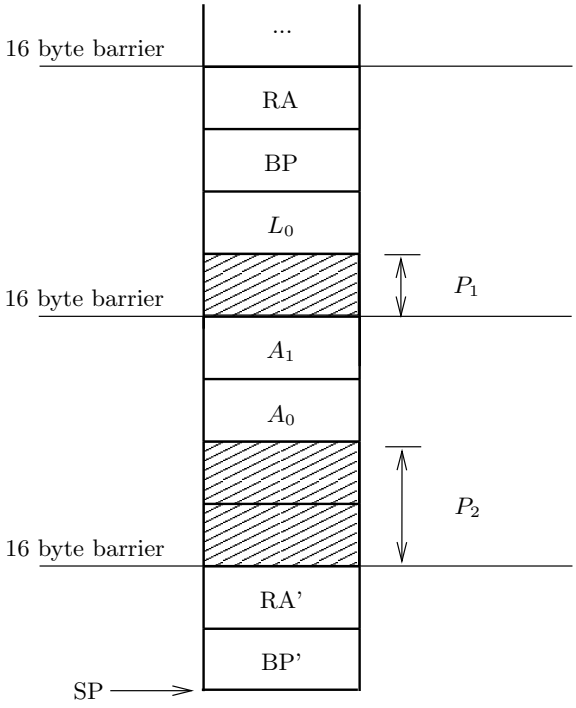


Fig. 2. Stack padding, as it can occur on Intel `ix86` platforms when GCC does not use a very high level of optimisation. L_0 is a local variable, A_1 and A_0 are function arguments.

It should be pointed out though, that, in recognition of portability requirements, all such platform-specific calculations were dropped, so that any computer user could test Super Sibcalls and record their responses to the patch via the GCC mailing lists. That is, the stack frame offset calculations were not necessarily needed in order to merely demonstrate the concept for integer values, for instance. However, if Super Sibcalls were to deal with long floating point arguments or big C structures to be passed via the stack, then simply relocating arguments would cause great problems. On a per-platform basis smaller data types would have to be fitted into large stack space segments and big data types, in turn, broken into many smaller pieces, according to the alignment requirements of GCC and the corresponding ABI.

The rest of the Super Sibcall patch is rather straightforward: at the end of RTL generation, in `sibcall.c` the zero-stack-frame-offset constraint must be bypassed. In this case, it is "safe" to do so, because, by deploying the aforementioned extensions of the parser, the front end (or programmer) assures the back end that no locals are live at this point.

4.2.3 Super Sibcalls vs. Normal Sibcalls

At first, Super Sibcalls appeared to be ideal to overcome the unpredictable problems of overlapping stack arguments. During the process of reviewing, however, subtle difficulties emerged, e. g. without extending the C grammar, how could the back end know whether a non-negative stack frame offset is merely due to temporaries, rather than storage to keep local variables? In order to address such issues, further workarounds dealing with the promotion and representation of temporaries in the back end would have to be examined.

Another weak spot of Super Sibcalls appears on more recent, especially 64-bit architectures, deploying modern calling conventions. This weakness involves the fact that a lot of processors are designed to pass arguments purely in registers, if possible. That is, even if Super Sibcalls were used on such systems, the arguments would still overlap, because the code addresses passing-on-stack, rather than passing-in-registers. To properly handle registers in Super Sibcalls, it would be necessary to introduce a generic algorithm for detecting which registers could be temporarily used for parameter shuffling. Normal sibling calls, however, perform equally for stack and register parameter passing: if it fails it drops *any* further optimisation attempts on this particular call.

So, even if Super Sibcalls could be implemented in a way that they perform well on many platforms, they do not bear the potential to substitute normal sibling calls. Passing in registers is too important, and disabling tail call optimisation per se for all such architectures is likely to be regarded as an inferior solution.

Even though, Super Sibcalls were not developed further in order to pursue other approaches as well which were more likely to integrate into GCC, they raised an interesting question: "Would it be suitable to improve tail call support

in GCC by offering an alternative mechanism for those platforms able to deploy it?" After all, the normal sibling calls would still work as expected.

4.3 CREATING A NEW CALLING CONVENTION

When thinking about the tail call problem of GCC, it is probably the most tempting of all ideas to come up with a wholly new calling convention and fade out concerns about other standards, at least temporarily. For reasons outlined throughout the previous chapters of this work, the C Calling Convention is simply not suited to support proper tail calls and recursion, and most of the constraints come down to the "caller deallocates" (function arguments) mechanism of C.

Chapter § 3 has already outlined several aspects which would have to be taken into account when designing such a new, custom calling convention. Foremost, it must be possible to have mutually recursive tail calls optimised between functions whose signature varies, for example:

```
float foo (int a, int b)
{
  /* Declare a 'big' C struct.  */
  struct big_c_struct local_struct;
  ...
  if (condition)
    return bar (local_struct);
  else
    return b;
}

float bar (struct big_c_struct arg)
{
  ...
  return foo (arg.value_1, arg.value_2);
}
```

In other words, proper tail calls must be a generally applicable concept, rather than an optimisation depending on technicalities of ABIs and CPUs. Therefore, with disregard to all prior C language and hardware peculiarities, the according epilogue instructions (for a run time stack growing downwards) should include the following steps:

1. Compute stack space required to store outgoing arguments.
2. Restore callee saved registers, if required.
3. Shift down return address, if |outgoing arguments| > |incoming arguments|.
4. If a base pointer is present, restore base pointer and adjust stack accordingly.

5. Issue jump to callee.

More or less, that is all that is required, as far as the epilogue is concerned. It could be argued as to whether 3. should be extended to the case where the return address has to be shifted up, instead of down. This would be an interesting move whenever the outgoing arguments require far less space than the incoming ones. However, that could be considered as fine tuning, while appending stack slots to the "bottom" would be absolutely essential.

In a second step, the designer of the calling convention has to decide whether all C functions use it alike, or if it is feasible to explicitly annotate tail calls, e. g. already in the source code. Both scenarios are possible, but get closer to the question about *binary compatibility*. Annotating the declaration, for example, would indicate that a certain function is removing its own arguments from the stack, but every other function still relies on the caller for that matter. While this works well, for example, with `stdcall` which basically resembles the widespread calling convention for the Pascal programming language, it would, using a custom convention, introduce an incompatibility when foreign C compilers attempt to link against a properly tail recursive subroutine.

The next bigger obstacle is, again, portability: a fully general solution, whether it is based on annotation or *complete* employment of a new convention, must be easy to port. That is, the alternative epilogue must also work on machines using register windows, for example, such as Sun Sparc computers. The GNU-C standard is very unlikely to be expanded with a calling convention that works on platform *XYZ*, but fails on all the others. Plus, it defeats much of the purpose of GHC using GCC as a portability layer.

Furthermore, an alternative calling convention for C has to deal with the usual features of the language, like variable argument functions and indirect calls. Unfortunately though (or luckily — depending on the point of view), in GCC some of these features depend solely on the machine-dependent code, rather than on RTL or `tree` code handling. In other words, even with a new calling convention in place, it would *not* be possible to automatically support indirect calls, for example.

4.4 SUMMARY

In this work, Super Sibcalls were the first approach to GNU C's tail call problem which were also implemented in parts. Because of the relatively huge overhead of a clean implementation, for relatively little gain, i. e. improvements in terms of overlapping stack arguments on Intel `ix86` architectures, the prototype did not get developed any further.

Discussions on the mailing lists have revealed weaknesses in Super Sibcalls that could only be addressed by supporting them in the machine-dependent parts of GCC and by porting the foundations for these changes to all platforms alike, such as the handling of floating point and structure arguments.

Considering the fact that Super Sibcalls solve the overlapping-argument problem by disabling GCC's existing code to marshal values without allocating stack

space, it was also very questionable at the time, whether the GCC community would accept such changes at all, even if an implementation would not have been as "ugly" as the prototype was. On top of it all, Super Sibcalls did not have the potential to solve GHC's most urgent problems with GCC as its back end (indirect tail calls).

A lot of time was also spent trying to design a calling convention which would be compatible to existing C programs, portable and also possible to implement for a single person in the time frame allowed for a "Diplomarbeit" (the German equivalent to a Master's Thesis). While it may have been possible to implement such a calling convention for (say) Intel platforms, there would certainly not have been enough time to transfer it on to others. Actually, an incomplete, new calling convention, bears four major problems:

- It would never be accepted in the CVS tree of GCC.
- It would not be very useful to GHC, because Haskell programmers are not necessarily bound to (say) Intel platforms, if...
- ...no predictions can be made whether the calling convention is portable at all.
- It would still miss GHC's most urgent GCC-related pains (consider indirect sibling calls, for instance).

In summary, a new calling convention is a bad idea under the given circumstances, because designing and prototyping the concepts which have been outlined in the current chapter, would have been a worthwhile academic exercise, but it would totally miss the point if the results are not in good enough shape or even too incomplete to be adopted by the GCC project.[3]

So, in order to support the cause of this thesis, an entirely different approach was chosen instead, not related to Super Sibcalls nor to the introduction of a new calling convention. However, it is an approach that has, indeed, made it into the official GCC sources and is definitely aimed at GHC's back end problems.

[3] After all, this is not the first thesis dealing with this problem (see Probst [2001], Nenzén and Rågård [2000]) and it can be seriously doubted whether more solutions are needed that work really well in theory but, for some unknown reason, gather dust in a university's library.

CHAPTER FIVE

IMPROVING GNU C

In GNU C development, as in most other open source projects, a contribution is not valued by its comprehensiveness alone, but by the way how new ideas are incorporated into a certain piece of code. If a (too) large patch is sent to the mailing list it often gets "ignored", simply because people lack the time to properly review the code and to find out how the lengthy changes interact with different parts of the compiler. Nonetheless, GCC is progressing fast, but it is taking only small steps while, at the same time, its maintainers make sure that no one steps on each other's heels.

Therefore, this chapter deals with the design, implementation, maintenance and also the process of contributing a whole series of *smaller* patches to improve the existing tail call support of the GCC suite. Apparently, all of the source code changes discussed in this chapter, have already been adopted by the maintainers and have also undergone several months of intense, successful testing.

The individual steps of the implementation are outlined in great detail with the intention of enabling other developers to create straightforward ports of the presented concepts and ideas to other platforms.

5.1 INTRODUCTION

Discussing an idea, or a prototype for an idea on the GCC mailing list is often very helpful to make the right decision on how to implement a feature, or to see how much work is effectively required to accomplish a certain task. Usually, people give helpful hints on how to improve things, or state their honest opinion if something should be approached from a rather different angle.

However, getting an actual piece of source code accepted by the maintainers of GCC requires more than just that (see also § 2). The *reviewing process* can take a long time and often results in having to rewrite a patch several times, i. e. presenting different versions of the very same piece of code multiple times on the mailing list.

The changes discussed in the following sections did undergo this process quite successfully, but in a distributed project like GCC it is also possible that a certain patch "falls through", because the developers with CVS write access have to set different priorities at a time (as it commonly happens before official program releases and the like) or, for example, go on holidays with their families

and friends. (People sometimes tend to forget that open source projects are also developed by real people, with real jobs, and who have real private lives, too.)

In the case of this thesis, however, nothing "fell through" and, fortunately, the early adoption of the most important source code changes did not only allow sufficient time for testing, but also for fixing some bugs.

5.2 A PRAGMATIC APPROACH

> The most important single aspect of software development is to be clear about what you are trying to build.
>
> — Bjarne Stroustrup, *The C++ Programming Language* (2000)

In § 3, the most stringent restrictions of the current tail call optimisation were presented; and in § 4, solutions were discussed that have the potential to solve some of the involved problems but would, unfortunately, introduce new burdens and obstacles. Hence, a new and more pragmatic optimisation approach had to be chosen that would allow smooth integration into GCC, on the one hand side, but would not cause new incompatibilities, on the other.

De facto, general support for indirect tail (or sibling) calls is not possible, because the calling conventions on platforms like ARM do not explicitly support or allow them. However, at the beginning of this work, in a meeting with the author, Prof. Simon Peyton Jones expressed how important those are, because the GHC generated C source code uses mostly indirect calls, and if he can not have the "£100 solution" (fully general), he would at least be interested in a "£10 solution" (indirect tail calls) which is properly supported by GCC, and which is not just part of the appendices of yet another thesis, and therefore, in the long run, not usable by his compiler.

For many different reasons outlined earlier in this work, a new calling convention itself did not offer the prospect of help under the circumstances, neither did Super Sibcalls, despite the promising name. So, it had to be examined if it is possible to introduce support for indirect sibling calls (which turns out to be a highly platform-specific feature) into GCC without breaking the platform-independent parts at the same time. In other words, is it possible to make a "£10" solution work on (say) Intel, without affecting ARM and similar systems?

The simple answer to this probing question is: yes, it is. GCC offers a rather poorly documented, yet powerful feature called "target hook" to handle such tasks. A target hook holds functionality which is specific to a certain target, without affecting others that do not support it. This is possible by defining a "worst case" value or a fall back function as default, and allow improvements whenever possible. For example, the functions to compute the costs of instruction scheduling and RTL code optimisation are partially implemented as a target hook. Each "machine" provides an independent way to determine the costs and maps the according evaluation functions to several different macros inside `machine.c`; the following snippet is taken from `i386.c`; it shows how a

universal hook (`TARGET_SCHED_ADJUST_COST`) gets assigned *individual* functionality (`ix86_adjust_cost`):

```
#undef TARGET_SCHED_ADJUST_COST
#define TARGET_SCHED_ADJUST_COST ix86_adjust_cost
```

Due to the fact that a target hook always defaults to at least *some* usable value or function definition, it is a very powerful concept of GCC that allows not only careful "fine tuning" of parameters and code optimisation, but also offers great portability, because new platforms do not necessarily have to have their own definitions for each hook in place; and if a certain hook is not suitable at all, it can simply be ignored without facing further consequences.

However, there is also a down side to this mechanism: target hooks can not be added arbitrarily, because they potentially slow down the compilation process as a whole and shift decisions concerning code generation deeper into the back end. That is, hooks should *only* be added when no other mechanism seems suitable. (GCC has become a very mature project, so introducing new hooks is not a very frequent procedure and, usually, is only performed by those who already maintain ports or bigger parts of GCC; hence, the lack of documentation for this process.)

5.3 INTRODUCING A NEW TARGET HOOK

Obviously, indirect sibling calls are predestined to be turned into a target hook, because the only reason why GCC originally refused to optimise those is that there is no sane way of supporting them in a platform-independent manner.

This thesis introduced `TARGET_FUNCTION_OK_FOR_SIBCALL` to give all platforms the possibility to either support indirect sibling calls, or to ignore them. Normally, the macro defaults to `false`, but it is up to the programmer to extend it with broader definitions and conditions as they seem fit for a particular platform.

Since target hooks are a very important feature of GCC, but are not explained in the documentation to an extent that would allow others to understand their implementation, this section describes the most essential steps which are necessary to introduce a new hook. Basically, these can be summarised as follows:

1. Declaring a new hook.
2. Defining a default.
3. Connecting the hook.
4. Modifying each target description.

Over the next pages, each of these steps is described in greater detail, trying to turn this process into a "purely mechanical procedure" for future hooks to be added.

Step 1. Implementation should always begin with the file `target.h` which describes a data structure called "`gcc_target`". Inside, further structures are defined, such as `sched` which is used for hooks related to instruction scheduling and general cost calculations. Sibling calls, however, do not need to be nested any further and thus, can be added directly at the end of `gcc_target`:

```
bool
(*function_ok_for_sibcall) PARAMS ((tree decl, tree exp));
```

This instruction declares a function (upon which the actual hook will be based) called "`function_ok_for_sibcall`". It accepts a `tree` declaration and expression as arguments; the former pointing to the function's declaration, should it be available.[1]

Step 2. The file `hooks.c` contains generic hooks which can be used as defaults by *hook initialisers*. A hook initialiser can be considered as the actual connection between the hook and a value, function, or predicate. For this work, the following "default function" had to be introduced:

```
bool
hook_tree_tree_bool_false (a, b)
     tree a ATTRIBUTE_UNUSED;
     tree b ATTRIBUTE_UNUSED;
{
  return false;
}
```

Obviously, `hook_tree_tree_bool_false` does not serve a specific purpose other than accepting two arguments of type `tree` and returning the value `false`. However, the advantage of such a general definition is that further, future hooks are able to use it as their fall back mechanism as well, should they also depend on two `tree`-type arguments.

The corresponding function declaration needs to be appended to the file `hooks.h`, so that the compiler knows about it at all. The code for that can be found in Appendix C.

Step 3. The hook needs to be initialised in order to use the function which was defined in Step 1 as a default. This happens in the file `target-def.h`, where the macro `TARGET_FUNCTION_OK_FOR_SIBCALL` first gets "connected" to an actual function declaration:

```
#define
TARGET_FUNCTION_OK_FOR_SIBCALL hook_tree_tree_bool_false
```

[1] A declaration is available if the call is direct. If a call is indirect, the macro `TREE_TYPE` has to be applied on the expression node in order to trace the callee's return type, number of arguments, and other properties (see also Fig. 3, p. 28).

In other words, whenever the macro `TARGET_FUNCTION_OK_FOR_SIBCALL` is accessed anywhere inside GCC, it will default to `false`, unless specifically defined otherwise.

The main initialisation, however, is invoked by attaching the new hook to another macro called "`TARGET_INITIALIZER`". It consists of the names of all deployed target hooks:

```
#define TARGET_INITIALIZER                      \
{                                               \
  TARGET_ASM_OUT,                               \
  TARGET_SCHED,                                 \
  TARGET_MERGE_DECL_ATTRIBUTES,                 \
  TARGET_MERGE_TYPE_ATTRIBUTES,                 \
  ...
  TARGET_CANNOT_MODIFY_JUMPS_P,                 \
  TARGET_FUNCTION_OK_FOR_SIBCALL,               \
  TARGET_IN_SMALL_DATA_P,                       \
  TARGET_BINDS_LOCAL_P,                         \
  ...
}
```

Step 4. In a final step, the hook gets tied to the actual targets, i. e. each target that is able to support (indirect) sibling calls must be given the possibility to overwrite the default function `hook_tree_tree_bool_false`. Commonly, this is done by adding two preprocessor instructions to the file `machine.c` (or `machine-protos.h`, should a static definition be required):

```
#undef TARGET_FUNCTION_OK_FOR_SIBCALL
#define
TARGET_FUNCTION_OK_FOR_SIBCALL machine_function_ok_for_sibcall
```

The first directive disables the hook's default; the second ties it to another custom function with the name "`machine_function_ok_for_sibcall`", where `machine` is the target name (see § 2 for an explanation of the GCC naming conventions).

5.4 EXTENDING THE MACHINE DESCRIPTION

Introducing the new hook was the "easy" part, and, at this stage, it is important to know how it works, but this step itself does not as yet solve any *real* problems. The main concern, enabling indirect sibling calls, has yet to be addressed, on each platform individually.

Even though the methodology discussed in this section is not meant to be a step-by-step manual, it can be divided, roughly, into the following parts, each described in great detail, not only to document the work involved, but also to present a comprehensive guideline for future ports:

- Converting the (old) macro.
- Improving the (new) hook.
- Adjusting the 32-bit call patterns.
- Adjusting the 64-bit call patterns.
- Using the (new) hook.

Since most of this work was accomplished on a 32-bit Intel platform, the text of this section will usually refer to the situation as it is (and was) on `i386` systems, unless specifically denoted otherwise. However, changes on the Intel platform are only meant as a reference implementation. All of the presented concepts can be transferred to other architectures, as this chapter will show. As a matter of fact, it is this high degree of flexibility which is the strong point of this work's solution.

5.4.1 Initial Situation

Before `TARGET_FUNCTION_OK_FOR_SIBCALL` existed, a target description contained several different macros, indicating whether a target supported, for example, an alternative sibling call epilogue, or sibling calls to position independent code, as well as other features. These macros were typically "rather simple" (semantically, not syntactically), because most of the decisions on successful sibling call optimisation had to be made in the platform-independent part of GCC's back end. For example, the following macro from the file `i386.h` was formerly used as a predicate on 32-bit Intel platforms to examine whether it was possible to optimise a tail call to a subroutine defined by its declaration `DECL`:

```
#define FUNCTION_OK_FOR_SIBCALL(DECL)                                    \
  ((DECL)                                                                \
   && (! flag_pic || ! TREE_PUBLIC (DECL))                               \
   && (! TARGET_FLOAT_RETURNS_IN_80387                                   \
       || ! FLOAT_MODE_P (TYPE_MODE (TREE_TYPE (TREE_TYPE (DECL)))) \
       || FLOAT_MODE_P (TYPE_MODE (TREE_TYPE (TREE_TYPE              \
                                              (cfun->decl))))))
```

Basically, this piece of code contains the constraints described in § 3.3, with the first line "`((DECL)`", being the most stringent one in terms of this thesis' subject; it basically means that the only parameter of this macro, the function declaration, must not be undefined, i. e. the function has to be called *directly*, but not via a pointer.

5.4.2 Converting the Macro

In an intermediate step, in order to get the new target hook working, the "old" macro should be merely converted *without* further extensions. In fact, this step turned out to be a necessity, because the GCC maintainers refused to have all of the changes incorporated into a single patch. The conversion itself is straightforward though and results in this new function definition:

```
static bool
ix86_function_ok_for_sibcall (decl, exp)
     tree decl;
     tree exp ATTRIBUTE_UNUSED;
{
  /* This is the previous FUNCTION_OK_FOR_SIBCALL macro.  */
  return ((decl)
          && (! flag_pic || ! TREE_PUBLIC (decl))
          && (! TARGET_FLOAT_RETURNS_IN_80387
              || ! FLOAT_MODE_P (TYPE_MODE (TREE_TYPE
                                            (TREE_TYPE (decl))))
              || FLOAT_MODE_P (TYPE_MODE (TREE_TYPE (TREE_TYPE
                                              (cfun->decl))))));
}
```

Declaring an attribute "unused" means that the interface for the new macro is in place, but that the corresponding function yet ignores the additional parameter. All the other instructions may (and, in most cases, should) remain unchanged by this process.

5.4.3 Improving the Hook

`ix86_function_ok_for_sibcall`, in the present state, is no more useful than `FUNCTION_OK_FOR_SIBCALL` originally was, because at this point it lacks the necessary improvements which would allow indirect sibling calls. Surprisingly enough, the main improvements to the macro do not require huge source code changes, if one is aware of the way GCC uses its `machine.c` files: simply removing the check for a sane function declaration is basically all that is required. However, without any further precautions this would result in (almost) all systems which have relations to the 32-bit Intel platform in one way or another "suddenly" supporting indirect sibling calls which is, of course, too optimistic.

It is so, because the 64-bit Intel-based systems may deploy different calling conventions which could hinder indirect sibling calls by demanding all call clobbered registers to be used for argument passing, for instance. In the case of AMD's "Hammer" this was not a problem though, because the authors of its calling convention are, in fact, all maintainers of GCC (see Hubicka et al. [2002]). The conventions for Intel's "Itanium", on the other hand, appear to be so fundamentally different that GCC has to support this architecture with independent machine descriptions. Even though this imposes difficulties in terms of maintenance, it was an advantage for this work, because all changes, made inside the `i386` directory of GCC's source tree, were isolated from the Itanium port.

The macro `TARGET_64BIT` checks whether GCC compiles code for a 64-bit architecture. Temporarily using it, when removing the "no indirect calls" check, leads to the desired bypass of indirect sibling calls on only the 64-bit platforms:

```
static bool
ix86_function_ok_for_sibcall (decl, exp)
     tree decl;
     tree exp;
{
  /* 64-bit architectures may not use indirect sibling
     calls.  */
  if (TARGET_64BIT && !decl)
    return false;

  /* If we are generating position-independent code, we cannot
     sibcall optimize any indirect call, or a direct call to a
     global function, as the PLT requires %ebx be live.  */
  if (!TARGET_64BIT && flag_pic
                    && (!decl || TREE_PUBLIC (decl)))
    return false;
  ...
  /* Otherwise okay.  That also includes certain types of
     indirect calls.  */
  return true;
}
```

5.4.4 Adjusting the 32-bit Call Patterns

At a first look, the improvements described in § 5.4.3 may seem peculiar, because it is not obvious at all *why* the 64-bit platforms can not deploy the same mechanisms and features as their predecessors, even though their properties are defined in the same set of GCC input files. However, the reason for this strict *logical* dissociation is that the introduction of a new and more liberal hook is not alone sufficient for a family of systems to actually support indirect sibling calls in terms of code generation; because, a hook merely tells the front end about the availability of a feature, not how it can actually be used to emit more efficient code for a platform.

As explained in § 2.2.4, a complete machine description consists of at least three files: `machine.h` and `machine.c` to describe platform properties and `machine.md` to match RTL instructions with actual assembly templates. Different platforms from the same family of architectures (called "`family`") may share `family.c` and `family.h` to generate RTL code, but may still rely on alternative `.md` files containing individual templates for emitting custom machine language. This is exactly what is happening with descriptions for the Intel processor family: the systems share most of the code, responsible for handling RTL, but require different patterns to translate it into assembly instructions within GCC's final pass, because the number of registers is varying, or the way they are used during a call, etc. Therefore, indirect calls need to be integrated into a target system's *call pattern* definition as well, not just in the C files. However, it should be clear, that extending the call patterns inside the `.md` files requires

a very good understanding of the according processor and the corresponding calling convention, because even the smallest flaw in just a single pattern could lead to fundamentally broken programs.

Table 1. IMPORTANT 32-BIT CALL PATTERNS

Pattern	Function Call is	Return Type
`call_0`	direct	`void`
`call_1`	indirect	`void`
`call_value_0`	direct	*value**
`call_value_1`	indirect	*value**
`call_pop_0`	direct	`struct`
`call_pop_1`	indirect	`struct`

* "*value*" means that any value may be returned except for a C `struct`, because returning a `struct` requires a different call mechanism (see 3.3 `e`).

Table 1 shows the most important call patterns of the `i386.md` file and sketches their purpose. The deployed naming scheme should be rather obvious: if the pattern ends with "`1`", it means that the call is indirect, otherwise it is direct. Furthermore, the patterns differentiate between making a call to a function that returns a value, a `struct`, or nothing at all (`void`).

The alert reader, however, may have noticed that the table in its current state, seems to lack *any* specific sibling call supporting patterns at all, even though GCC has been optimising those for several years already, in cases where the function call in the tail position is direct and the typical constraints are fulfilled. Actually, this is not a bug, it is a feature: until now, no special call patterns were necessary, because direct sibling calls can make use of what is already there, like `call_value_0` or `call_0`, for instance. As explained in § 3, no extra (call clobbered) register is needed when making a direct call, regardless of whether it is in the tail position or not. Hence, the existing call patterns could be applied without any further changes in order to issue at least the direct sibling calls.

In order to support indirect sibling calls for 32-bit Intel-based systems, however, two new patterns need to be added to this list: `sibcall_1` for indirect sibling calls to `void` functions, and `sibcall_value_1` for indirect sibling calls to functions which return a value other than a C `struct`.

The call pattern `call_1` originally looked like this, which should come as a surprise, after all that has been said about sibling call support in GCC (or the lack thereof):

```
(define_insn "*call_1"
 [(call (mem:QI (match_operand:SI 0 "call_insn_operand" "rsm"))
        (match_operand 1 "" ""))]
 "!TARGET_64BIT"
{
  if (constant_call_address_operand (operands[0], QImode))
    {
```

```
        if (SIBLING_CALL_P (insn))
          return "jmp\t%P0";
        else
          return "call\t%P0";
      }
    if (SIBLING_CALL_P (insn))
      return "jmp\t%A0";
    else
      return "call\t%A0";
  }
    [(set_attr "type" "call")])
```

The pattern does, indeed, address indirect sibling calls (`SIBLING_CALL_P`), even though GCC did not allow those on a much higher level of code generation.[2] In other words: during RTL generation, GCC used to disable all indirect sibling calls while, further down the back end, it had the concepts in place that were meant to support them — theoretically.

However, the *correct* way to handle this, in regard to the new target hook, is to split the pattern into two: `call_1` would then *only* handle indirect calls which are not in the tail position of a function, and the new `sibcall_1` gets assigned the responsibility for indirect sibling calls. Figure 1 shows the differing conditions (see double framed text) which clearly separate the task of each pattern and indicate that those are meant to work only on 32-bit platforms. The differentiation is very important, because of the new constraints (see framed text) in `sibcall_1` that demand the call's destination address to be either available as immediate integer operand, or to be stored in register `%ecx`, `%edx`, or `%eax`. Of course, this constraint is not portable at all, because another platform, even if it is (historically) related to 32-bit Intel, most certainly has a different number and also usage of registers.

It should become clear now, why the original definition of `call_1` was broken (that is, if GCC would have passed indirect sibling calls on to the level of machine code generation): the pattern's constraints were yet not appropriate, i. e. did not impose restrictions on proper register usage. The old and general constraint "`rsm`" basically means that the operand can either be *any* general purpose register, an immediate integer, or a memory location with any kind of address that the machine generally supports. That is, the pattern did not take into account at all that not all registers are equally well suited to carry a machine's "jump target". In many cases, and by pure coincidence, the pattern may have been assigned the "right" register; in many cases, however, the register allocator may have come up with the wrong decision, leading to unpredictable program crashes.

Although comprehensive, the above work for 32-bit systems is only half complete. That is, the process needs to be repeated in almost exactly the same

[2] The author of this text has to admit that he does not know whether this is merely a relict from the past, a test case for a particular purpose, or whether the code is just plain wrong.

```
(define_insn "*call_1"
 [(call (mem:QI (match_operand:SI 0 "call_insn_operand" "rsm"))
        (match_operand 1 "" ""))]
 "!SIBLING_CALL_P (insn) && !TARGET_64BIT"
{
  if (constant_call_address_operand (operands[0], QImode))
    return "call\t%P0";
  return "call\t%A0";
}
  [(set_attr "type" "call")])


(define_insn "*sibcall_1"
 [(call (mem:QI (match_operand:SI 0 "call_insn_operand" "s,c,d,a"))
        (match_operand 1 "" ""))]
 "SIBLING_CALL_P (insn) && !TARGET_64BIT"
{
  if (constant_call_address_operand (operands[0], QImode))
    return "jmp\t%P0";
  return "jmp\t%A0";
}
  [(set_attr "type" "call")])
```

Fig. 1. The *old* call pattern `call_1` has been split into a revised `call_1` and a new `sibcall_1` pattern.

manner for the pattern `sibcall_value_1`, in order to support indirect calls to functions returning a value. Albeit, the concepts and techniques presented here can be transferred straightforward which means there is no need to wade through this process in greater detail; instead, the corresponding source code changes can be found in Appendix C.

5.4.5 Adjusting the 64-bit Call Patterns

The restrictions inside call patterns like `call_1` point up to the need for specialised 64-bit versions. It is not possible to implement a new feature, which involves changes in GCC's call patterns, only for 32-bit systems if the 64-bit successor depends on the availability of the according pattern changes just as much. In consequence, there are two possible options for implementing such a feature: 1. create *one* pattern which is able to cover both platforms at the same time, or 2. create a pattern `pattern_x_rex64` for each pattern `pattern_x`. In order to implement indirect sibling calls for the "Intel family", the second approach must be chosen, especially in light of all previous changes to the 32-bit patterns.

Table 2 shows the most important call patterns GCC originally offered to support 64-bit systems based upon the `x86-64` machine description. Even though the table does not include great surprises, it is interesting to see that there is no custom pattern for `call_0`. The reason for this has already been outlined in

Table 2. IMPORTANT CALL PATTERNS FOR 64-BIT

Pattern	Function Call is	Return Type
`call_0`	direct	`void`
`call_1_rex64`	indirect	`void`
`call_value_0_rex64`	direct	*value*
`call_value_1_rex64`	indirect	*value*

the previous paragraph: a direct (sibling) call does not demand any special constraints imposed upon register usage, as can be clearly seen from the relatively simple pattern code:

```
(define_insn "*call_0"
  [(call (mem:QI (match_operand 0
                 "constant_call_address_operand" ""))
         (match_operand 1 "" ""))]
  ""
{
  if (SIBLING_CALL_P (insn))
    return "jmp\t%P0";
  else
    return "call\t%P0";
}
  [(set_attr "type" "call")])
```

Without any stringent checks, the patterns treat "normal" and sibling calls almost identically, choosing the built in `jmp` command over `call`, if the RTL instruction is, indeed, labelled as sibling call.

What needs to be added to the machine description is an equivalent of `sibcall_1`, namely `sibcall_1_rex64`. This pattern is intended to be used when the RTL instructions for indirect sibling calls are matched onto machine code templates on a 64-bit Intel-based platform, like `x86-64`. For various internal optimisation and performance related issues which are beyond the scope of this thesis, `sibcall_1_rex64` is actually divided into two parts, `sibcall_1_rex64` and `sibcall_1_rex64_v`:

```
(define_insn "*sibcall_1_rex64"
  [(call (mem:QI (match_operand:DI 0
                 "constant_call_address_operand" ""))
         (match_operand 1 "" ""))]
  "SIBLING_CALL_P (insn) && TARGET_64BIT"
  "jmp\t%P0"
  [(set_attr "type" "call")])

(define_insn "*sibcall_1_rex64_v"
  [(call (mem:QI (reg:DI 40))
```

```
        (match_operand 0 "" ""))]
  "SIBLING_CALL_P (insn) && TARGET_64BIT"
  "jmp\t*%%r11"
  [(set_attr "type" "call")])
```

The patterns appear to be fairly simple due to the usage of register classes for 64-bit systems, i. e. rather than explicitly suggesting registers that can be potentially used for an indirect sibling call, an entire *class* of available registers is already predefined and used by those patterns containing the suffix `rex64`. The additional `v` indicates that a special volatile, call clobbered register, which is not part of the standard class, may be used to hold the call's operand as well; this increases flexibility.

Another noticeable difference is that the template generally matches 64-bit `DI`-mode integer operands which are eight times the size of the smallest addressable unit `QI` which is only 8 bits long. In contrast, Fig. 1 shows how the 32-bit patterns handle only 32-bit `SI`-mode integer operands. Even though, this is a obvious and logical difference, it is a noteworthy detail and especially important for other developers trying to port the patterns because, at this stage, GCC does not really know about `int` or `float` types anymore, rather than about units which are mapped to C types, and are summarised in the following table [Stallman 2002]:

`QI`: An integer that is as wide as the smallest addressable unit, usually 8 bits.

`HI`: An integer, twice as wide as a `QI`-mode integer, usually 16 bits.

`SI`: An integer, four times as wide as a `QI`-mode integer, usually 32 bits.

`DI`: An integer, eight times as wide as a `QI`-mode integer, usually 64 bits.

`SF`: A floating point value, as wide as a `SI`-mode integer, usually 32 bits.

`DF`: A floating point value, as wide as a `DI`-mode integer, usually 64 bits.

The reason why the 64-bit call pattern(s) are designed the way they are goes beyond the scope of this work though. Originally, the author of this text suggested how these could be implemented, but several months of testing on the `x86-64` platform (and the corresponding cross compiler) showed various, sometimes hard to detect weaknesses, either in the quality of the generated code or in terms of compile time when, for instance, additional or modified register classes were added.

Today's version of the patterns was implemented and continuously tested, mainly by the same (SuSE) engineers who also helped design the ABI for `x86-64` [Hubicka et al. 2002] and who happen to host some of the biggest automated test suites available for GCC (see also § 2.1).

Just as it was the case with 32-bit call pattern changes, the description of the counterpart `sibcall_1_value_rex64` will be omitted, because it is rather similar to implementing `sibcall_1_rex64` and can also be found in full length in Appendix C.

5.4.6 Using the Hook

Finally, with both the new 32 and 64-bit call patterns in place, the target hook `ix86_function_ok_for_sibcall` can be extended to accept also indirect function calls on `x86-64` platforms. Considering the code excerpt on page 70 again, all that is necessary is removing the check for `TARGET_64BIT` at the beginning, which results in this new function definition:

```
static bool
ix86_function_ok_for_sibcall (decl, exp)
     tree decl;
     tree exp;
{
  /* If we are generating position-independent code, we cannot
     sibcall optimize any indirect call, or a direct call to a
     global function, as the PLT requires %ebx be live.  */
  if (!TARGET_64BIT && flag_pic
                    && (!decl || TREE_PUBLIC (decl)))
    return false;
  ...
  /* Otherwise okay.  That also includes certain types of
     indirect calls.  */
  return true;
}
```

The new hook and the according call pattern improvements can be used by telling the RTL generating front end about it. Inside `calls.c`, where most of the important decisions concerning sibling call optimisation are made (see § 3), the previous constraints imposed by the following condition

```
01   if ( ...
02        || fndecl == NULL_TREE
03        || (flags & (ECF_RETURNS_TWICE | ECF_LONGJMP))
04        || TREE_THIS_VOLATILE (fndecl)
05        || !FUNCTION_OK_FOR_SIBCALL (fndecl) ... )
06      try_tail_call = 0;
```

can be replaced with the more elegant and also flexible solution:

```
01   if ( ...
02        || !(*targetm.function_ok_for_sibcall) (fndecl, exp)
03        /* Functions that do not return exactly once
04           may not be sibcall optimized.  */
05        || (flags &
06            (ECF_RETURNS_TWICE | ECF_LONGJMP | ECF_NORETURN))
07        || TYPE_VOLATILE
08              (TREE_TYPE
09                 (TREE_TYPE
10                    (TREE_OPERAND (exp, 0)))) ... )
11       try_tail_call = 0;
```

In other words, `calls.c` does not depend on a valid function declaration anymore (lines 7–10: the expression node `exp` is sufficient and is the only link to the callee, should it be invoked via a pointer) and it lets other parts of the back end make a decision on whether the call should be optimised or not (line 2). Hence, with this last minor modification, 32 and 64-bit Intel-based systems both fully support indirect sibling calls. Other platforms which are, in theory, not bound by their ABI to *not* support this notion, can be extended in a similar way, if one has sufficient information on the use and availability of call clobbered registers on the corresponding target.

CHAPTER SIX

RESULTS

The code changes discussed in § 5 aim to support indirect sibling calls in mainline GCC. The impact, however, might not be obvious yet and so some results and real world examples are presented to demonstrate how developers and their programs may benefit from this work. Hence, this chapter will explain the new GCC features by examining various source code snippets and the corresponding machine code on 32 and 64-bit Intel platforms.

6.1 TECHNICAL IMPACT

> Rule: It is a mistake to use the manufacturer's special call instruction.
>
> — Ken Thompson, *Plan 9: The Early Papers*
> *A New C Compiler* (1990)

The improved sibling call optimisation will be officially available when GCC 3.4 is released, and can be already tested by downloading the contents of the `basic-improvements` CVS branch. The following examples have all been compiled using a current snapshot of this branch and will, of course, lead to fundamentally different results if tested with *any* older version of the compiler.

Example 1.

```
extern int bar (int);
int (*ptr) (int);

int foo (int a)
{
  ptr = bar;
  return a? (*ptr) (a) : 0;
}
```

The code has to be compiled using the command line switches `-O1 -dp -S -foptimize-sibling-calls` to obtain an annotated assembler output which contains the names of the deployed instruction patterns. On an Intel-based 32-bit platform, GCC will produce the following output:

```
foo:
        pushl   %ebp            # 46    *pushsi2
        movl    %esp, %ebp      # 47    *movsi_1/2
        movl    8(%ebp), %eax   # 3     *movsi_1/1
        testl   %eax, %eax      # 11    *cmpsi_ccno_1/1
        je      .L2             # 12    *jcc_1
        movl    ptr, %ecx       # 16    *movsi_1/2
        popl    %ebp            # 53    popsi1
        jmp     *%ecx           # 17    *sibcall_value_1/2
.L2:
        movl    $0, %eax        # 35    *movsi_1/1
        popl    %ebp            # 50    popsi1
        ret                     # 51    return_internal
```

The interesting parts are underlined: instead of using the processor's built in `call` command to issue a function call to `bar`, GCC optimises the indirect tail call into a sibling call by using the new pattern `sibcall_value_1` to jump to the desired subroutine.

In Example 1 the improvements are obvious: by jumping to the subroutine, instead of using the built in `call` instruction, `bar` does not have to allocate a new stack frame; and `foo`'s incoming argument space can be "recycled".

Another noticeable property of this example is the jump's target: `%ecx`. Instead of jumping straight to a function label, the pattern has to store the address in a register, predestined by `sibcall_value_1`.

In comparison, the assembler output looks like this, if compiled without sibling call optimisation or, alternatively, with a version of GCC prior to 3.4:

```
foo:
        pushl   %ebp            # 37    *pushsi2
        movl    %esp, %ebp      # 38    *movsi_1/2
        subl    $8, %esp        # 39    *pro_epilogue_
                                            adjust_stack_1/1
        movl    8(%ebp), %edx   # 3     *movsi_1/2
        movl    $0, %eax        # 21    *movsi_1/1
        testl   %edx, %edx      # 11    *cmpsi_ccno_1/1
        je      .L1             # 12    *jcc_1
        movl    %edx, (%esp)    # 13    *movsi_1/4
        call    *ptr            # 15    *call_value_1
.L1:
        movl    %ebp, %esp      # 42    *pro_epilogue_
                                            adjust_stack_1/2
        popl    %ebp            # 43    popsi1
        ret                     # 44    return_internal
```

Clearly, this produces less efficient code, because a) the `call` is more expensive than a `jmp`, b) `foo` does not really return when it calls `bar` and c), because `bar` will have to allocate its own stack frame. Hence, Example 1 is (asymptotically) faster and consumes no extra memory when issuing control to the callee.

Example 2. As pointed out in § 3, sibling calls are not restricted to functions sharing a common signature. The improvements, made to the sibling call optimisation do not constrict this mechanism. Hence, the following example is also subject to optimisation on all platforms which use the same byte representation for integer and long integer values:

```
int (*ptr) (int);
... /* Assign *ptr here somewhere.  */
long foo (long a)
{
  ...
  return ( (*ptr) ((int) a) );
}
```

It is, however, not "clean" in a sense that the return value should really be explicitly casted, but for the purpose of demonstrating a concept it does not really matter. Albeit, in `i386` platforms, the following machine code is generated from the example:

```
foo:
        pushl   %ebp            # 47    *pushsi2
        movl    %esp, %ebp      # 48    *movsi_1/2
        subl    $4, %esp        # 49    *pro_epilogue_
                                            adjust_stack_1/1
        fnstcw  -2(%ebp)        # 41    x86_fnstcw_1
        movzwl  -2(%ebp), %eax  # 42    *movhi_1/4
        fldl    8(%ebp)         # 45    *movdf_nointeger/1
        orw     $3072, %ax      # 43    *iorhi_1/1
        movw    %ax, -4(%ebp)   # 44    *movhi_1/6
        fldcw   -4(%ebp)        # 14    fix_truncsi_memory
        fistpl  8(%ebp)
        fldcw   -2(%ebp)
        movl    ptr, %ecx       # 15    *movsi_1/2
        leave                   # 51    leave
        jmp     *%ecx           # 16    *sibcall_value_1/2
```

On `x86-64`, where `int` corresponds to 4 bytes and `long` to 8, this is *not* possible (see also Hubicka et al. [2002]). However, if both `foo` and `bar` are integer returning functions, one obtains the following machine code fragment:

```
foo:
        cvttsd2si  %xmm0, %edi      # 13  fix_truncdfsi_sse
        movq       ptr(%rip), %r11  # 17  *movdi_1_rex64/2
        jmp        *%r11            # 18  *sibcall_value_1_rex64_v
```

Example 3. This last example is one which demonstrates a case where the improved GCC correctly disregards sibling call optimisation. As outlined in § 3, a function which does not return, or is marked as being volatile, must not be sibcall optimised:

```
typedef void no_return_func (void);
no_return_func (*ptr) __attribute__((noreturn));

int foo (void)
{
  (*ptr) ();
}
```

Correctly, on 32-bit Intel platforms, GCC produces the following machine code making use of the standard call mechanism by deploying the low level pattern `call_1`:

```
foo:
        pushl   %ebp            # 23  *pushsi2
        movl    %esp, %ebp      # 24  *movsi_1/2
        subl    $8, %esp        # 25  *pro_epilogue_adjust_stack_1/1
        call    *ptr            # 9   *call_1
```

What is shown in Example 3 is not self evident, because when issuing an indirect call (in the tail position), GCC can not use the target function's declaration. Instead, similar to Fig. 3, shown on page 28, the expression node has to be "traversed" to find out whether the callee eventually returns.

6.2 A NEW TEST SUITE FOR SIBCALLS

A new feature with a substantial influence on GCC's code generation abilities should not be adopted without a corresponding test suite in place. As mentioned in § 2.1, DejaGnu is the preferred test environment for GCC. That is, inside the `testsuite/` directory of the source tree, hundreds of DejaGnu input files can be found, either for rather broad "stress tests", or for special features and correlating problem report numbers (on the mailing lists, commonly referred to as PRs).

During this work, the maintainer of GCC's `MMIX` port [Nilsson 2001] has started building a systematic sibling call test suite by contributing five different DejaGnu input files mainly addressing recursion. These files are automatically called by many automated test systems employed by companies like SuSE or Red Hat and show up regressions, promptly.

Since this work's new sibling call mechanism was accepted for GCC, it was another aim of this thesis to make sure it is maintainable by any third party and that it can be addressed by today's test systems in the usual way. In other words, potential sibling call port maintainers must be automatically notified, in case their efforts do not pass the most common sibling call use and test cases.

This work introduced a new DejaGnu test which targets specifically indirect sibling calls. Basically, the input file is straightforward C, but contains special annotations to instruct DejaGnu *how* it should treat the example:

```
/* { dg-do run { target i?86-*-* x86_64-*-*} } */
/* { dg-options "-O2 -foptimize-sibling-calls" } */
```

```
int foo (int);
int bar (int);

int (*ptr) (int);
int *f_addr;

int
main ()
{
  ptr = bar;
  foo (7);
  exit (0);
}

int
bar (b)
     int b;
{
  if (f_addr == (int*) __builtin_return_address (0))
    return b;
  else
    abort ();
}

int
foo (f)
     int f;
{
  f_addr = (int*) __builtin_return_address (0);
  return (*ptr)(f);
}
```

The first two lines are commented out, so the C compiler does not complain, but they do contain valuable DejaGnu information: `dg-do run` means that the source code should be run on targets whose description matches the template `i?86-*-* x86_64-*-*`, and the `dg-options` set the appropriate compiler switches.

If the example exits with error status 0, the test is passed. If `abort` was called, it failed. However, the interesting part of the test is the custom function `__builtin_return_address` which gives back the return address of the current function, or of one of its callers. Its argument is the number of frames to scan up the call stack. A value of 0 yields the return address of the current function, a value of 1 yields the return address of the caller of the current function, and so forth.

With sibling calls, the callee does not create a new stack frame, hence a sibling call shares the return address with its caller. Therefore, function `foo` stores the

return address in a global variable `f_addr` and lets `bar` compare its value with its own current return address. If they match, the test was successful, otherwise it would mean that a new stack was opened, indeed (or something else went terribly wrong).

Alternatively, a test could have been written, using the directive `dg-do compile` and then the assembler output would be checked for the keyword `jmp`. This solution, however, is not portable, but it provides yet another example of DejaGnu's flexibility.

The presented test has become an integrated part of GCC's test suite and can also be found in full length in Appendix C. It can be invoked manually (assuming the file name `sibcall-6.c`) by using the following directive:

```
$ make check-gcc RUNTESTFLAGS="dg.exp=sibcall-6.c"
```

It is also possible to run the entire test suite manually by omitting the variable definition for `RUNTESTFLAGS`. Further information on using the entire test suite and selected tests are also available in Stallman [2002] and on the GCC homepage (see Appendix B).

6.3 A PRACTICAL GUIDE TO WORKING ON GCC

Working on GCC can be a somewhat Bohemian experience, when only having the common notions and terms of "traditional" software engineering in mind: in GCC, there are no Waterfalls or Spirals, and its design often results from immediate necessity, rather than from a cost benefit analysis and careful forward planning. In some ways this has bitten the developers and users of GCC alike (e. g. when the C++ ABI had to be replaced with a new and incompatible *cross vendor ABI* [Intel Corporation 2001], in version 3.0 of GCC), but over nearly the last two decades, the community has benefited from GCC's unconventional ways of getting things done. The high number of supported platforms and its many users second that and are a testimony to GCC's broad acceptance and success.

It was yet another aim of this thesis to outline the process of getting involved in an unusual project like GCC which does not really know about a single project leader, a quality assurance department, and flashy (introductory) manuals. So, where to start?

Documentation. Surprisingly enough, the work involved for this thesis did *not* begin with an extensive study of the GCC documentation, even though it should be pointed out that the *GNU Compiler Collection Internals* [Stallman 2002] provided valuable information on the various source files involved in each pass of the compiler. At least this rather basic information is necessary to be even able to ask the right questions on the mailing lists and to find out which parts of GCC's code base may be subject of change during the work. Hence, the most crucial hurdle to overcome was understanding how the sources work and interact. As a matter of fact, there are many references and symbols in

GCC's documentation which may not even make sense, unless one is deeply familiar with the source code already.

This basically means that there is no other way to become involved except by reading and making sense of GCC's source code. However, for an experienced programmer this should not impose huge difficulties, because very often a piece of code is easier to understand than a lengthy (and not always accurate) description in prose.

As a rule of thumb, an upcoming contributor should at least read the files `tree.*` and `rtl.*`, because they contain the definition and accessor macros for GCC's intermediate program representation. Major parts of the GCC documentation are nothing else but a summary of these two files and will be easier to comprehend given the right background information.

> Good code is its own best documentation. As you're about to add a comment, ask yourself, 'How can I improve the code so that this comment isn't needed?' Improve the code and then document it to make it even clearer.
>
> — Steve McConnell, *Code Complete: A Practical Handbook of Software Construction* (1993)

Creating good patches. Getting comfortable with GCC's internals and the way it is being developed was, of course, inevitable for this work and did cause — no doubt — initial difficulties. The real problem (despite finding the "right" approach to GCC's tail call problem at all), however, was to integrate the necessary changes into *compact* patch files which would be accepted by the maintainers.

As can be seen in Appendix C, all the required changes were fed to the list in "small packets", each addressing only one problem at a time. What is more, each packet required a lengthy discussion with various experienced GCC developers who would suggest improvements and point out mistakes. Hence, a patch had not only to be written, but also maintained over a long period of time in which the underlying GCC code base keeps changing. Admittedly, this can be a rather stressing and frustrating process at times, but it is very rewarding once an e-mail from a maintainer is received that says: "Applied." (Some of the patches for this work took longer than two months from their initial posting until they have been considered fit to be applied.)

Maintaining a patch means keeping up to date with current development, but also with the ongoing discussions on the lists. Thus, good communication skills are not only required in an office environment, but also in a distributed project like GCC (even, if on a different level only), because the other developers are not obliged to help, and receiving a detailed code review should be mostly regarded as a privilege rather than a "right". But exactly here lies another crucial point which has to be fully understood: no one will review different versions of one and the same patch over and over again, if the rewritten patches are not progressing into a "sensible direction". In other words, creating good patches is usually an evolutionary process which must result in measurable improvements to be successful.

Then again, new code can also get rejected, either because it is not explained well enough in its comments, or because misleading terminology issued. The GCC community is very much concerned about the readability of their code and the corresponding comments, because often the comments are not only the best but also the only documentation available for a specific feature.

> The price of correct programming is eternal vigilance.
>
> — Rob Pike
> (personal communication, Dec. 7, 2002)

Bug Fixing. Maintenance for a patch does not end upon adoption, i. e. people will get back to the author of a piece of code, in case the automatic regression tests report problems. In fact, it sometimes happens that an already committed patch gets removed again, just because it introduced a problem which is too difficult to fix for others while the original author is on vacation, for example.

Indeed, this work introduced two minor bugs which would cause many of the bigger test suites available for GCC to fail. One of the glitches caused by the author's patch was due to a typo which was correctly addressed by fellow developer Zack Weinberg (see Fig. 1).

```
Index: config/i386/i386.c
--- config/i386/i386.c  17 Oct 2002 15:43:22 -0000       1.447.2.13
+++ config/i386/i386.c  20 Oct 2002 18:10:56 -0000
@@ -1377,7 +1377,7 @@ ix86_function_ok_for_sibcall (decl, exp)
   /* If we are generating position-independent code, we cannot sibcall
      optimize any indirect call, or a direct call to a global function,
      as the PLT requires %ebx be live.  */
-  if (flag_pic && (!decl || !TREE_PUBLIC (decl)))
+  if (flag_pic && (!decl || TREE_PUBLIC (decl)))
     return false;

   /* If we are returning floats on the 80387 register stack, we cannot
```

Fig. 1. This patch was sent to the GCC mailing list to remove a bug introduced by a typo (underlined).

Another mistake was fixed by the author himself, even before the various test suites were given the chance to complain: the conversion from working with a function's `tree` declaration to using the expression node, made it necessary to access the callee's arguments and return types via linked `struct` members, rather than directly. In other words, when dealing with indirect calls it is not possible to check, for example, whether the `volatile` bit of the declaration is set; instead, the macro `TREE_TYPE` has to be recursively applied on the expression node to reach the `FUNCTION_TYPE`.[1] Figure 2 contains the brief patch to clean up this misconception.

[1] A little bit of extra information may be necessary at this point: if the callee is marked being volatile, it does not mean that the function pointer is as well. Therefore, it is wrong to examine only the function's pointer object.

```
Index: calls.c
--- calls.c     1 Oct 2002 20:22:34 -0000               1.231.4.7
+++ calls.c     7 Oct 2002 03:49:59 -0000
@@ -2444,7 +2444,7 @@ expand_call (exp, target, ignore)
       || !(*targetm.function_ok_for_sibcall) (fndecl, exp)
       || (flags & (ECF_RETURNS_TWICE | ECF_LONGJMP))
       /* Functions that do not return may not be sibcall optimized. */
-      || TYPE_VOLATILE (TREE_TYPE (TREE_OPERAND (exp, 0)))
+      || TYPE_VOLATILE (TREE_TYPE (TREE_TYPE (TREE_OPERAND (exp, 0))))
       /* If this function requires more stack slots than the current
          function, we cannot change it into a sibling call.  */
       || args_size.constant > current_function_args_size
```

Fig. 2. Functions marked as being volatile may not be sibcall optimised; this patch fixes a bug where GCC would check a `POINTER_TYPE` for the `volatile` bit instead of its `TREE_TYPE`, which is a `FUNCTION_TYPE`.

These two examples show how important it is to create readable and therefore maintainable code. Even though, the author of a patch is made responsible for any impacts it has — good or bad — code should always be designed in a way that it allows others to find and remove bugs in it as well.

Summary. Although, GCC is a breathtakingly large software development project, it is not impossible to get involved in it, even in such a short period of time, as is the case when writing a "Diplomarbeit". The lack of introductory manuals and a "professional helpdesk" is compensated by consistent code design and informative comments in the code.

It is a necessity to study certain source files, before even getting started in development (the fact that the `tree` definitions exceed 3000 lines of code does not make the task any easier), because descriptions and manuals can not replace the information given in the sources themselves. After all, these are what the GCC developer aims to modify.

In the very early beginning of this thesis, and due to the complexity of GCC, the thought arose, whether it would be a worthwhile undertaking to approach the main (tail call) problem independent of an already existing compiler; but, the results of this thesis strengthen the argument that the improvements gained are far greater than they would be, should someone try to write his own optimising C back end in only six months time.

A few pieces of advice which this chapter could give to the GCC beginner are as follows:

- *"Get the hands dirty"* (i. e. start to code) as quickly as possible: people should not spend too much time trying to figure out manuals. It all becomes clear (even the manuals!), once people are more familiar with GCC's basic data structures and the sources in general.

- *"Keep it simple."* This is an old UNIX philosophy [Kernighan and Pike 1984, § 2] which also translates to GCC as one of its most widespread compilers. A patch that adds a feature by sacrificing maintainability or bloating the code base is unlikely to find acceptance in the community. If help is required to redesign something, the mailing lists are a formidable entry point to find assistance in these matters.

- *Testing, testing, and even more testing.* Patches sent to the list are expected to be thoroughly tested by the contributor. In fact, each patch should go with a detailed description on how this new piece of code was tested. The usual methods include bootstrapping (a cross compiler) and using the internal DejaGnu stress test suite.

- *The involved paper work must not be underestimated!* As pointed out in § 2, all contributors must sign a Copyright Agreement with the Free Software Foundation (FSF), before their work can be adopted. Depending on how busy the FSF is, this may take quite some weeks, sometimes even months.

The list could be continued, but further concepts come in naturally once the initial obstacles are out of the way. The intention of this chapter was to give people a couple of simple to follow guidelines on how to approach a huge open source project like GCC and to show how and why it may differ from conventional closed source undertakings. From this perspective, four simple rules should be sufficient, at least to get started.

CHAPTER SEVEN

CONCLUSIONS

Due to the results of this thesis, mainstream GCC is now able to offer support for indirect sibling calls on Intel-based platforms, such as `i?86` and `x86-64`. The implementation is flexible enough to be ported to PowerPC, Alpha, or Sparc-based systems (amongst others) simply by extending the corresponding GCC machine descriptions. However, the idea for this particular approach did not arise until late, when GCC's various sibcall constraints had been carefully analysed (see § 3) and other potential solutions had been abandoned due to shortcomings which could not be solved, given the time constraints imposed by a "Diplomarbeit" (see § 4).

7.1 CURRENT STATE

Prof. Simon Peyton Jones now has an integrated and well maintained "£10 solution" (see § 5.1) in the GNU compiler suite which brings the Evil Mangler closer to the end of its life time. Despite previous approaches and hopes to dispose this script earlier (see Fig. 1), GCC versions prior to 3.4 did not even offer the prospect of achieving this goal, because the front end GHC depends especially on the optimisation of indirect function calls. With the lack of such an important optimisation feature, additional annotations and compiler switches alone do not possess the power and flexibility to make up for it.

However, even with the achievements of this work, it is still unlikely that the mangler script can yet be removed for good. GCC still imposes many constraints on the sibcall optimisation stage of the compiler; these resemble those that have already been examined in § 3 of this work. Unfortunately, one of the main reasons why it is so difficult to overcome these constraints is, at the same time, one of GCC's biggest strengths: the compiler suite makes a great cross compiler for more than 200 software and hardware platforms [Pizka 1997], which means that new features have to be implemented in a mostly platform-independent manner to be supported properly (see § 2.2). If a certain functionality is not compatible with a system's ABI, or the way its underlying processor operates, it often means that this particular feature can not be made available to *any* platform, because it would not be possible to represent it properly in the machine-independent part of the back end. Indirect sibling

```
From: Duncan Coutts <dcoutts@cray.com>
Date: Tue, 21 Aug 2001 11:34:23 -0500
Subject: end of the 'evil mangler' ?

Hi all,

Are we ever likely to see the end of the 'evil mangler'?
With gcc 3.x and it's selection of __atribute__ annotations (eg
noreturn, pure, const, naked etc) would we be able to iliminate the
need for the mangler?

http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_5.html#SEC92

What kind of annotations would gcc need before we could dump the
mangler?

Duncan

Ps. this is all just wishful thinking, as I'm currently faced with the
prospect of modifying the EM.
http://www.cse.unsw.edu.au/~chak/haskell/ghc/comm/the-beast/
mangler.html
```

Fig. 1. This message was sent to the `glasgow-haskell-users` list upon availability of GCC version 3.0. (Message taken from the mailing list archive, `http://www.haskell.org/pipermail/glasgow-haskell-users/2001-August/002211.html`.)

calls are a good example for this notion, but also show that there are, indeed, work arounds, like the introduction of a new target hook as it is described in § 5.3.

In other words, this work made indirect sibling calls potentially available to the vast majority of supported GCC targets, but "incompatible systems" like the ARM-based ones are bound to miss out due to the design shortcomings in their own ABI.

7.2 FUTURE WORK

Due to the flexibility of the presented solution, without doubt, a lot of future work will go into porting the GCC changes on to other platforms. While the author wrote down the results of his thesis, several people have already raised interest to help with this task in regard to the platforms Sparc, Alpha and PowerPC.

Furthermore, it is also worth thinking about concepts that would help, in particular, lifting the "zero stack frame" constraint, because many GCC front ends are, and would be, able to do a liveness analysis themselves, hence knew about the cases in which it is safe to optimise a tail call, even if there are local variables stored on the current stack frame. As a matter of fact, the

author's Super Sibcall patch does contain code which adds another keyword, "`__tailcall`", to GCC's parser module to let a front end or a programmer enforce optimisation (see § 4.2) for these cases.

Related to that, it would also be very interesting to find out whether it is practical to implement an extended scope or liveness analysis in a C compiler back end and to examine how optimisation could benefit from that as a whole. Chapters § 1.5 and § 3.3 have already outlined sibcall constraints which could be bypassed with such a mechanism in place, but it has yet to be examined how it would integrate into a mature and portable compiler such as GCC.

C is certainly not designed to prevent the programmer or a custom language front end from doing something stupid with pointers; so additional precautions could prove useful, even if they would not reach the level of sophistication as, for example, Java's *garbage collection* did, which relies strongly on liveness and scope information associated with program variables. In fact, a concept like garbage collection, which frees memory allocated by objects not referenced or needed any longer, has already been proposed by Baker [1995] to address GCC's tail call problem. Baker proposed a solution where he would allow the run time stack to nearly overflow, before he systematically removes old stack frames. The advantage of this approach is that neither the C epilogue, nor the prologue require any modifications as they still open a new stack frame for each and every call; the downside, however, would be that this method is bound to fail for general purpose C programs where tail calls and normal calls can occur at the same frequency, because for normal calls the stack frames must not be destroyed. Asymptotically, in such cases, the garbage collection would always collect at the very end of the run time stack, not being able to remove a sufficient number of frames. When Baker proposed his idea, he had a front end for the language Scheme in mind, but he did not suggest it would also be general enough to be applicable for all C code alike. This, however, is a point that played a significant role for this work and for almost all GNU C developers alike because, foremost, GNU C should serve as a reliable and greatly portable *C* compiler.

7.3 RESUMÉ

Even though this work brought GCC one important step closer to a solution of the tail call problem, it is not yet fully solved. However, the extensions can already be used to improve compilation of functional programming languages, but can have a noticeable impact on imperative programs as well. For example, it is now possible to elegantly implement a simple state machine in C, where each function represents a state in the state machine and the state transition is an indirect function call in the tail position. Obviously, such an application of continuation passing is, therefore, not only useful for (say) the Haskell community.

For the author, this thesis has also been very important, personally, because it lead to a very deep commitment and interest in this not entirely new, but,

nonetheless, exciting field of computer science. With the skills and knowledge gained during this work, he will be able to contribute source code to the GCC project in the future as well. This aspect is an especially rewarding one, because GCC happens to be one of those prestigious open source projects which is normally rather picky about its contributors. Of course, due to the General Public License, everyone is welcome and allowed to modify GCC as they please but sometimes, it can be very difficult to convince the community to adopt one's changes, especially if the maintainers have no reason to "trust" a contributor, yet.

APPENDIX A

COMPILER FOUNDATIONS

THE DEFINITIONS presented in this chapter are done so for the convenience of readers who are not yet fully comfortable with terminology deployed in the field of compiler construction. Every (good) text book ought to cover these expressions as well, but rather than making only references to the literature the author considered it a good idea to add the relevant terms to this work, so they are all in one place.

A.1 ACTIVATION RECORD

The *Activation Record* is a data structure which holds information needed by a program's function in execution. Its values include:

- return value
- parameters
- optionally: a control link
- optionally: an access link
- saved machine status (registers, etc.)
- local data
- temporaries

Typically, an Activation Record is stored on the run time stack by allocating a contiguous block of memory, called "*stack frame*". Not all compilers and programming languages use the exact same stack frame layout, because sometimes registers can take place of one or more of its fields [Aho et al. 1986].

The return value points to the calling function; the parameters are a function's incoming arguments, and the optional links fields are usually references to other stack frames in order to access foreign variables, for instance. Saving a machine's state usually refers to pushing representative registers on the stack; and temporaries are necessary, if the compiler needs space to marshal arguments, or register values. The function's local variables are stored in the area reserved for local data.

A.2 BASIC BLOCK ANALYSIS

Aho et al. [1986, § 9.4] describe a *basic block* as follows: "A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end." Basic blocks are a very important concept in the field of compiler construction, because they are the cornerstone for building directed *flow graphs*.

The analysis of an input program's control flow is important for many optimisation and transformation techniques, and especially when trying to determine whether an instruction (e. g. a call) is in a function's tail position, or not. Figure 1 shows a brief, but typical flow graph consisting of two basic blocks, B_1 and B_2:

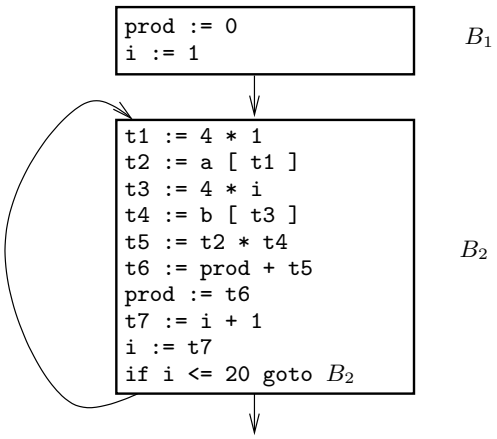
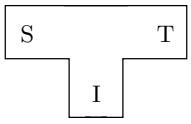


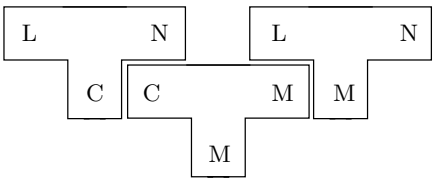
Fig. 1. A typical flow graph for a program [Aho et al. 1986, § 9.4].

A.3 BOOTSTRAPPING

A compiler is characterised by three languages: the source language S that it translates, the target (language) T that it generates code for, and the implementation language I that it is written in. This is commonly represented with a so called *T-diagram* and can also be textually expressed as $\mathrm{S}_{\mathrm{I}}\mathrm{T}$:



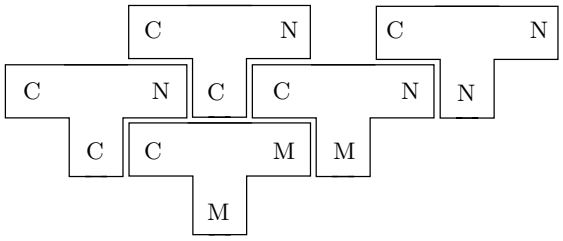
Hence, one way of obtaining a C-written *cross compiler* for language L and foreign target N on a machine M, which can be programmed using machine code M, can be depicted as follows:



$\mathrm{L}_{\mathrm{C}}\mathrm{N}$ is the compiler implementation written in C. It should be obvious from the diagram that the working, native C-compiler $\mathrm{C}_{\mathrm{M}}\mathrm{M}$ is the chief prerequisite for creating the new cross compiler $\mathrm{L}_{\mathrm{M}}\mathrm{N}$.

This is where the concept of *bootstrapping* comes in, because if this process is applied multiple times for one and the same language, the result will be a compiler which is basically implemented in the same language it is supposed to translate. Therefore, bootstrapping can be thought of as the ability of a programming language to compile itself [Aho et al. 1986, § 11.2].

Example. In this example, bootstrapping of a C compiler for language (or platform) N is depicted. The main implementation $\mathrm{C}_{\mathrm{C}}\mathrm{N}$ is first used to obtain $\mathrm{C}_{\mathrm{M}}\mathrm{N}$ and then *again* to obtain the final result $\mathrm{C}_{\mathrm{N}}\mathrm{N}$:



The main point is that, bootstrapping is extremely useful for a wide variety of different areas. For example, it can not only be deployed to implement a compiler for language L^{+} in L^{+} itself, but it can also be used to successively bring up an optimising compiler for L^{+}, beginning with merely a small subset of it, (say) L^{-} (see Aho et al. [1986, § 11.2, ex. 11.3]); and, of course, bootstrapping proves useful to thoroughly test a compiler by feeding it with its very own code base as it commonly happens in a lot of the GCC tests.

APPENDIX B

INTERNET ADDRESSES

Binutils `<http://sources.redhat.com/binutils/>`
The GNU Binutils are a collection of binary tools. The main ones are an assembler and a linker to gain executable binary files on a system. The Binutils are, like other GNU software, free and open.

DDD `<http://www.gnu.org/software/ddd/>`
DDD is a graphical front end for command line debuggers such as GDB, DBX, WDB, Ladebug, JDB, XDB, the Perl debugger, or the Python debugger. Besides "usual" front end features such as viewing source texts, DDD has become popular through its interactive graphical data display, where data structures are displayed as graphs.

diet libc `<http://www.dietlibc.org/>`
The diet libc is a C system library that is optimised for small size. It can be used to create small statically linked binaries for Linux on `alpha`, `arm`, `hppa`, `ia64`, `i386`, `mips`, `s390`, `sparc`, `sparc64`, `ppc` and `x86_64`.

DejaGnu `<http://www.gnu.org/software/dejagnu/>`
DejaGnu is a free framework for testing other programs. Its purpose is to provide a single front end for all tests. It can be thought of as a custom library of Tcl procedures crafted to support writing a test harness. A test harness is the testing infrastructure that is created to support a specific program or tool.

Emacs `<http://www.gnu.org/software/emacs/>`
Emacs is the extensible, customisable, self documenting real time display editor, freely available for many different platforms. It offers content sensitive modes for a wide variety of file types, from plain text to source code to HTML files and is scriptable using the Lisp programming language.

Glasgow Haskell Compiler `<http://www.haskell.org/ghc/>`
The Glasgow Haskell Compiler is a robust, fully featured, optimising compiler and interactive environment for Haskell 98. Apart from documentation, it also freely provides a generational garbage collector, and a space and time profiler.

GNU C Library `<http://www.gnu.org/software/libc/>`
Any UNIX-like operating system needs a C library: the library which defines system calls and other basic facilities such as `open`, `malloc`, `printf`, `exit`, and so on. The GNU C library is used as the C library in the GNU system and most newer systems based upon the Linux kernel.

GNU Compiler Collection `<http://www.gnu.org/software/gcc/>`
The GNU Compiler Collection contains front ends for C, C++, Objective C, Fortran, Java, and Ada, as well as libraries for these languages. It is free and open to anyone who is interested in the project, or in getting a widespread and established cross platform compiler.

GNU Debugger `<http://sources.redhat.com/gdb/>`
GDB, the GNU Project debugger, allows one to see what is going on "inside" another program while it executes — or to see what another program was doing at the moment it crashed. Many people use DDD as a front end to GDB.

Mercury `<http://www.cs.mu.oz.au/research/mercury/>`
Mercury is a new logic/functional programming language, which combines the clarity and expressiveness of declarative programming with advanced static analysis and error detection features. Similar to GHC, compilation relies mainly on the GCC back end, due to the translation of Mercury code into either a high or low level C program.

Newlib `<http://sources.redhat.com/newlib/>`
Newlib is a C system library intended for use on embedded systems. It is a conglomeration of several library parts, all under free software licenses that make them easily usable on embedded products. It can be compiled for a wide array of processors, and will usually work on any architecture.

OpenOffice `<http://www.openoffice.org/>`
The OpenOffice suite is the open source predecessor of the commercial, Sun owned StarOffice. It contains a text processor similar to Microsoft Word, calculation software along the lines of Microsoft Excel, and many other programs used in everyday office work.

APPENDIX C

SOURCE CODE

THE VARIOUS PATCHES described in C.1 are now integral part of the GCC suite and have been created by the author of this thesis. Due to the copyright agreement with the Free Software Foundation, the source code is published under the terms of the GNU General Public License, version two or higher [Free Software Foundation 1991].

Each patch starts with an individual `ChangeLog` entry explaining what has been addressed and changed. Apart from being available in this appendix, the patches can also be obtained electronically, either by sending e-mail to the author, via download from `http://www.andreasbauer.org/`, or by putting a query into the searchable mailing list archives of GCC.

The source code of sections C.2 and C.3 is not part of the official CVS repository and, therefore, copyright by the author of this thesis.

C.1 INDIRECT SIBLING CALLS

The patches, enabling indirect sibling calls in GCC, have been applied to the `basic-improvements` CVS branch and will first appear in official releases of version 3.4 of GCC.

Patch 1: Turning `FUNCTION_OK_FOR_SIBCALL` into a Target Hook

```
01   2002-10-01  Andreas Bauer  <baueran@in.tum.de>
02
03           * calls.c (expand_call): Remove the 'no indirect check'
04           for sibcall optimization; use function_ok_for_sibcall
05           target hook; refine check for 'function is volatile'.
06           (FUNCTION_OK_FOR_SIBCALL): Remove the redefinition.
07           * hooks.c (hook_tree_tree_bool_false): New.
08           * hooks.h (hook_tree_tree_bool_false): Declare.
09           * target-def.h (TARGET_FUNCTION_OK_FOR_SIBCALL): New.
10           (TARGET_INITIALIZER): Add it.
11           * target.h (struct gcc_target): Add function_ok_for_sibcall.
12           * config/alpha/alpha.c: (alpha_function_ok_for_sibcall): New.
13           (TARGET_FUNCTION_OK_FOR_SIBCALL): Redefine accordingly.
14           * config/alpha/alpha.h: (FUNCTION_OK_FOR_SIBCALL): Remove.
15           * config/arm/arm-protos.h: (arm_function_ok_for_sibcall):
16           Remove function declaration.
17           * config/arm/arm.c: (arm_function_ok_for_sibcall): Make
```

```
18          function static and accept another argument of type ‘tree’.
19          (TARGET_FUNCTION_OK_FOR_SIBCALL): Redefine accordingly.
20          * config/arm/arm.h: (FUNCTION_OK_FOR_SIBCALL): Remove.
21          * config/frv/frv.h: (FUNCTION_OK_FOR_SIBCALL): Remove.
22          * config/i386/i386.c: (ix86_function_ok_for_sibcall): New.
23          (TARGET_FUNCTION_OK_FOR_SIBCALL): Redefine accordingly.
24          * config/i386/i386.h: (FUNCTION_OK_FOR_SIBCALL): Remove.
25          * config/pa/pa-linux.h: (FUNCTION_OK_FOR_SIBCALL): Remove.
26          (TARGET_HAS_STUBS_AND_ELF_SECTIONS): New definition.
27          * config/pa/pa.c: (pa_function_ok_for_sibcall): New.
28          (TARGET_FUNCTION_OK_FOR_SIBCALL): Redefine accordingly.
29          * config/pa/pa.h: (FUNCTION_OK_FOR_SIBCALL): Remove.
30          * config/rs6000/rs6000-protos.h: (function_ok_for_sibcall):
31          Remove function declaration.
32          * config/rs6000/rs6000.c: (rs6000_function_ok_for_sibcall):
33          Rename function_ok_for_sibcall to rs6000_function_ok_for_sibcall;
34          rename first argument to ‘decl’; accept another argument
35          of type ‘tree’; make static.
36          (TARGET_FUNCTION_OK_FOR_SIBCALL): Redefine accordingly.
37          * config/rs6000/rs6000.h: (FUNCTION_OK_FOR_SIBCALL): Remove.
38          * config/sh/sh.c: (sh_function_ok_for_sibcall): New.
39          (TARGET_FUNCTION_OK_FOR_SIBCALL): Redefine accordingly.
40          * config/sh/sh.h: (FUNCTION_OK_FOR_SIBCALL): Remove.
41          * config/sparc/sparc.c: (sparc_function_ok_for_sibcall): New.
42          (TARGET_FUNCTION_OK_FOR_SIBCALL): Redefine accordingly.
43          * config/sparc/sparc.h: (FUNCTION_OK_FOR_SIBCALL): Remove.
44          * config/xtensa/xtensa.h: (FUNCTION_OK_FOR_SIBCALL): Remove.
45
46  Index: calls.c
47  ===================================================================
48  RCS file: /cvsroot/gcc/gcc/gcc/calls.c,v
49  retrieving revision 1.231.4.5
50  diff -w -u -p -r1.231.4.5 calls.c
51  --- calls.c     20 Sep 2002 01:29:06 -0000      1.231.4.5
52  +++ calls.c     28 Sep 2002 05:00:02 -0000
53  @@ -36,10 +36,6 @@ Software Foundation, 59 Temple Place - S
54   #include "langhooks.h"
55   #include "target.h"
56
57  -#if !defined FUNCTION_OK_FOR_SIBCALL
58  -#define FUNCTION_OK_FOR_SIBCALL(DECL) 1
59  -#endif
60  -
61   /* Decide whether a function's arguments should be processed
62      from first to last or from last to first.
63
64  @@ -2443,17 +2439,12 @@ expand_call (exp, target, ignore)
65           It does not seem worth the effort since few optimizable
66           sibling calls will return a structure.  */
67         || structure_value_addr != NULL_RTX
68  -      /* If the register holding the address is a callee saved
69  -       register, then we lose.  We have no way to prevent that,
70  -       so we only allow calls to named functions.  */
71  -      /* ??? This could be done by having the insn constraints
72  -       use a register class that is all call-clobbered.  Any
73  -       reload insns generated to fix things up would appear
74  -       before the sibcall_epilogue.  */
```

```
75    -      || fndecl == NULL_TREE
76    +      /* Check whether the target is able to optimize the call
77    +        into a sibcall.  */
78    +      || !(*targetm.function_ok_for_sibcall) (fndecl, exp)
79           || (flags & (ECF_RETURNS_TWICE | ECF_LONGJMP))
80    -      || TREE_THIS_VOLATILE (fndecl)
81    -      || !FUNCTION_OK_FOR_SIBCALL (fndecl)
82    +      /* Functions that do not return may not be sibcall optimized.  */
83    +      || TYPE_VOLATILE (TREE_TYPE (TREE_OPERAND (exp, 0)))
84           /* If this function requires more stack slots than the current
85             function, we cannot change it into a sibling call.  */
86           || args_size.constant > current_function_args_size
87    Index: hooks.h
88    ===================================================================
89    RCS file: /cvsroot/gcc/gcc/gcc/hooks.h,v
90    retrieving revision 1.5
91    diff -w -u -p -r1.5 hooks.h
92    --- hooks.h     21 Aug 2002 02:41:44 -0000      1.5
93    +++ hooks.h     28 Sep 2002 05:00:03 -0000
94    @@ -27,5 +27,5 @@ bool hook_tree_bool_false PARAMS ((tree)
95     void hook_tree_int_void PARAMS ((tree, int));
96     void hook_void_void PARAMS ((void));
97     void hook_FILEptr_constcharptr_void PARAMS ((FILE *, const char *));
98    -
99    +bool hook_tree_tree_bool_false PARAMS ((tree, tree));
100    #endif
101   Index: hooks.c
102   ===================================================================
103   RCS file: /cvsroot/gcc/gcc/gcc/hooks.c,v
104   retrieving revision 1.5
105   diff -w -u -p -r1.5 hooks.c
106   --- hooks.c     21 Aug 2002 02:41:44 -0000      1.5
107   +++ hooks.c     28 Sep 2002 05:00:03 -0000
108   @@ -62,3 +62,12 @@ hook_FILEptr_constcharptr_void (a, b)
109         const char *b ATTRIBUTE_UNUSED;
110    {
111    }
112   +
113   +/* Hook that takes two trees and returns false.  */
114   +bool
115   +hook_tree_tree_bool_false (a, b)
116   +     tree a ATTRIBUTE_UNUSED;
117   +     tree b ATTRIBUTE_UNUSED;
118   +{
119   +  return false;
120   +}
121   Index: target.h
122   ===================================================================
123   RCS file: /cvsroot/gcc/gcc/gcc/target.h,v
124   retrieving revision 1.33.2.3
125   diff -w -u -p -r1.33.2.3 target.h
126   --- target.h    17 Sep 2002 22:58:47 -0000      1.33.2.3
127   +++ target.h    28 Sep 2002 05:00:04 -0000
128   @@ -244,6 +244,11 @@ struct gcc_target
129         not, at the current point in the compilation.  */
130      bool (* cannot_modify_jumps_p) PARAMS ((void));
131
```

```
132   +  /* True if it is OK to do sibling call optimization for the specified
133   +     call expression EXP.  DECL will be the called function, or NULL if
134   +     this is an indirect call.  */
135   +  bool (*function_ok_for_sibcall) PARAMS ((tree decl, tree exp));
136   +
137      /* True if EXP should be placed in a "small data" section.  */
138      bool (* in_small_data_p) PARAMS ((tree));
139
140   Index: target-def.h
141   ===================================================================
142   RCS file: /cvsroot/gcc/gcc/gcc/target-def.h,v
143   retrieving revision 1.30.2.3
144   diff -w -u -p -r1.30.2.3 target-def.h
145   --- target-def.h        17 Sep 2002 22:58:47 -0000      1.30.2.3
146   +++ target-def.h        28 Sep 2002 05:00:05 -0000
147   @@ -245,6 +245,7 @@ Foundation, 59 Temple Place - Suite 330,
148
149    /* In hook.c.  */
150    #define TARGET_CANNOT_MODIFY_JUMPS_P hook_void_bool_false
151   +#define TARGET_FUNCTION_OK_FOR_SIBCALL hook_tree_tree_bool_false
152
153    #ifndef TARGET_IN_SMALL_DATA_P
154    #define TARGET_IN_SMALL_DATA_P hook_tree_bool_false
155   @@ -271,6 +272,7 @@ Foundation, 59 Temple Place - Suite 330,
156      TARGET_EXPAND_BUILTIN,                       \
157      TARGET_SECTION_TYPE_FLAGS,                   \
158      TARGET_CANNOT_MODIFY_JUMPS_P,                        \
159   +  TARGET_FUNCTION_OK_FOR_SIBCALL,              \
160      TARGET_IN_SMALL_DATA_P,                      \
161      TARGET_BINDS_LOCAL_P,                                \
162      TARGET_ENCODE_SECTION_INFO,                  \
163   Index: config/alpha/alpha.c
164   ===================================================================
165   RCS file: /cvsroot/gcc/gcc/gcc/config/alpha/alpha.c,v
166   retrieving revision 1.272.2.3
167   diff -u -p -r1.272.2.3 alpha.c
168   --- config/alpha/alpha.c        20 Sep 2002 01:29:08 -0000      1.272.2.3
169   +++ config/alpha/alpha.c        1 Oct 2002 00:25:44 -0000
170   @@ -118,6 +118,8 @@ int alpha_this_literal_sequence_number;
171    int alpha_this_gpdisp_sequence_number;
172
173    /* Declarations of static functions.  */
174   +static bool alpha_function_ok_for_sibcall
175   +  PARAMS ((tree, tree));
176    static int tls_symbolic_operand_1
177      PARAMS ((rtx, enum machine_mode, int, int));
178    static enum tls_model tls_symbolic_operand_type
179   @@ -292,6 +294,9 @@ static void unicosmk_unique_section PARA
180    #undef  TARGET_EXPAND_BUILTIN
181    #define TARGET_EXPAND_BUILTIN alpha_expand_builtin
182
183   +#undef TARGET_FUNCTION_OK_FOR_SIBCALL
184   +#define TARGET_FUNCTION_OK_FOR_SIBCALL alpha_function_ok_for_sibcall
185   +
186    struct gcc_target targetm = TARGET_INITIALIZER;
187    ^L
188    /* Parse target option strings.  */
```

```
189  @@ -2267,6 +2272,19 @@ alpha_legitimize_address (x, scratch, mo
190
191       return plus_constant (x, low);
192     }
193  +}
194  +
195  +/* We do not allow indirect calls to be optimized into sibling calls, nor
196  +   can we allow a call to a function in a different compilation unit to
197  +   be optimized into a sibcall.  */
198  +static bool
199  +alpha_function_ok_for_sibcall (decl, exp)
200  +     tree decl;
201  +     tree exp ATTRIBUTE_UNUSED;
202  +{
203  +  return (decl
204  +          && (! TREE_PUBLIC (decl)
205  +              || (TREE_ASM_WRITTEN (decl) && (*targetm.binds_local_p) (decl))));
206   }
207
208   /* For TARGET_EXPLICIT_RELOCS, we don't obfuscate a SYMBOL_REF to a
209  Index: config/alpha/alpha.h
210  ===================================================================
211  RCS file: /cvsroot/gcc/gcc/gcc/config/alpha/alpha.h,v
212  retrieving revision 1.176.4.7
213  diff -u -p -r1.176.4.7 alpha.h
214  --- config/alpha/alpha.h        23 Sep 2002 04:38:44 -0000      1.176.4.7
215  +++ config/alpha/alpha.h        1 Oct 2002 00:25:52 -0000
216  @@ -1165,14 +1165,6 @@ extern int alpha_memory_latency;
217       }                                                                    \
218   }
219
220  -/* We do not allow indirect calls to be optimized into sibling calls, nor
221  -   can we allow a call to a function in a different compilation unit to
222  -   be optimized into a sibcall.  */
223  -#define FUNCTION_OK_FOR_SIBCALL(DECL)                   \
224  -  (DECL                                                 \
225  -   && (! TREE_PUBLIC (DECL)                             \
226  -       || (TREE_ASM_WRITTEN (DECL) && (*targetm.binds_local_p) (DECL))))
227  -
228   /* Try to output insns to set TARGET equal to the constant C if it can be
229      done in less than N insns.  Do all computations in MODE.  Returns the place
230      where the output has been placed if it can be done and the insns have been
231  Index: config/arm/arm-protos.h
232  ===================================================================
233  RCS file: /cvsroot/gcc/gcc/gcc/config/arm/arm-protos.h,v
234  retrieving revision 1.29.8.2
235  diff -u -p -r1.29.8.2 arm-protos.h
236  --- config/arm/arm-protos.h     17 Sep 2002 22:58:53 -0000      1.29.8.2
237  +++ config/arm/arm-protos.h     1 Oct 2002 00:25:54 -0000
238  @@ -41,7 +41,6 @@ extern unsigned int  arm_compute_initial
239   #ifdef TREE_CODE
240   extern int    arm_return_in_memory     PARAMS ((tree));
241   extern void   arm_encode_call_attribute        PARAMS ((tree, int));
242  -extern int    arm_function_ok_for_sibcall PARAMS ((tree));
243   #endif
244   #ifdef RTX_CODE
245   extern int    arm_hard_regno_mode_ok   PARAMS ((unsigned int, enum
```

```
246  machine_mode));
247  Index: config/arm/arm.c
248  ===================================================================
249  RCS file: /cvsroot/gcc/gcc/gcc/config/arm/arm.c,v
250  retrieving revision 1.223.2.5
251  diff -u -p -r1.223.2.5 arm.c
252  --- config/arm/arm.c    20 Sep 2002 01:29:09 -0000      1.223.2.5
253  +++ config/arm/arm.c    1 Oct 2002 00:26:28 -0000
254  @@ -117,6 +117,7 @@ static void  arm_set_default_type_attrib
255   static int      arm_adjust_cost                PARAMS ((rtx, rtx, rtx, int));
256   static int      count_insns_for_constant       PARAMS ((HOST_WIDE_INT, int));
257   static int      arm_get_strip_length           PARAMS ((int));
258  +static bool      arm_function_ok_for_sibcall    PARAMS ((tree, tree));
259   #ifdef OBJECT_FORMAT_ELF
260   static void     arm_elf_asm_named_section      PARAMS ((const char *, unsigned
261  int));
262   #endif
263  @@ -192,6 +193,9 @@ static void  arm_internal_label             PARAMS
264   #undef TARGET_ASM_INTERNAL_LABEL
265   #define TARGET_ASM_INTERNAL_LABEL arm_internal_label
266
267  +#undef TARGET_FUNCTION_OK_FOR_SIBCALL
268  +#define TARGET_FUNCTION_OK_FOR_SIBCALL arm_function_ok_for_sibcall
269  +
270   struct gcc_target targetm = TARGET_INITIALIZER;
271   ^L
272   /* Obstack for minipool constant handling.  */
273  @@ -2266,16 +2270,17 @@ arm_is_longcall_p (sym_ref, call_cookie,
274
275   /* Return nonzero if it is ok to make a tail-call to DECL.  */
276
277  -int
278  -arm_function_ok_for_sibcall (decl)
279  +static bool
280  +arm_function_ok_for_sibcall (decl, exp)
281        tree decl;
282  +     tree exp ATTRIBUTE_UNUSED;
283   {
284     int call_type = TARGET_LONG_CALLS ? CALL_LONG : CALL_NORMAL;
285
286     /* Never tailcall something for which we have no decl, or if we
287        are in Thumb mode.  */
288     if (decl == NULL || TARGET_THUMB)
289  -    return 0;
290  +    return false;
291
292     /* Get the calling method.  */
293     if (lookup_attribute ("short_call", TYPE_ATTRIBUTES (TREE_TYPE (decl))))
294  @@ -2287,20 +2292,20 @@ arm_function_ok_for_sibcall (decl)
295        a branch instruction.  However, if not compiling PIC, we know
296        we can reach the symbol if it is in this compilation unit.  */
297     if (call_type == CALL_LONG && (flag_pic || !TREE_ASM_WRITTEN (decl)))
298  -    return 0;
299  +    return false;
300
301     /* If we are interworking and the function is not declared static
302        then we can't tail-call it unless we know that it exists in this
```

```
303          compilation unit (since it might be a Thumb routine).  */
304        if (TARGET_INTERWORK && TREE_PUBLIC (decl) && !TREE_ASM_WRITTEN (decl))
305    -     return 0;
306    +     return false;
307
308        /* Never tailcall from an ISR routine - it needs a special exit sequence.  */
309        if (IS_INTERRUPT (arm_current_func_type ()))
310    -     return 0;
311    +     return false;
312
313        /* Everything else is ok.  */
314    -   return 1;
315    +   return true;
316     }
317
318     ^L
319    Index: config/arm/arm.h
320    ===================================================================
321    RCS file: /cvsroot/gcc/gcc/gcc/config/arm/arm.h,v
322    retrieving revision 1.155.4.5
323    diff -u -p -r1.155.4.5 arm.h
324    --- config/arm/arm.h    20 Sep 2002 01:29:10 -0000      1.155.4.5
325    +++ config/arm/arm.h    1 Oct 2002 00:26:40 -0000
326    @@ -1502,12 +1502,6 @@ typedef struct
327     #define FUNCTION_ARG_REGNO_P(REGNO)    (IN_RANGE ((REGNO), 0, 3))
328
329     ^L
330    -/* Tail calling.  */
331    -
332    -/* A C expression that evaluates to true if it is ok to perform a sibling
333    -   call to DECL.  */
334    -#define FUNCTION_OK_FOR_SIBCALL(DECL) arm_function_ok_for_sibcall ((DECL))
335    -
336     /* Perform any actions needed for a function that is receiving a variable
337        number of arguments.  CUM is as above.  MODE and TYPE are the mode and type
338        of the current parameter.  PRETEND_SIZE is a variable that should be set to
339    Index: config/frv/frv.h
340    ===================================================================
341    RCS file: /cvsroot/gcc/gcc/gcc/config/frv/frv.h,v
342    retrieving revision 1.3.2.5
343    diff -u -p -r1.3.2.5 frv.h
344    --- config/frv/frv.h    20 Sep 2002 01:29:12 -0000      1.3.2.5
345    +++ config/frv/frv.h    1 Oct 2002 00:27:28 -0000
346    @@ -3539,9 +3539,6 @@ frv_ifcvt_modify_multiple_tests (CE_INFO
347        scheduling.  */
348     #define FIRST_CYCLE_MULTIPASS_SCHEDULING_LOOKAHEAD frv_sched_lookahead
349
350    -/* Return true if a function is ok to be called as a sibcall.  */
351    -#define FUNCTION_OK_FOR_SIBCALL(DECL) 0
352    -
353     enum frv_builtins
354     {
355       FRV_BUILTIN_MAND,
356    Index: config/i386/i386.c
357    ===================================================================
358    RCS file: /cvsroot/gcc/gcc/gcc/config/i386/i386.c,v
359    retrieving revision 1.447.2.3
```

```
diff -u -p -r1.447.2.3 i386.c
--- config/i386/i386.c  17 Sep 2002 22:58:57 -0000      1.447.2.3
+++ config/i386/i386.c  1 Oct 2002 00:28:16 -0000
@@ -742,6 +742,7 @@ static int ix86_save_reg PARAMS ((unsign
 static void ix86_compute_frame_layout PARAMS ((struct ix86_frame *));
 static int ix86_comp_type_attributes PARAMS ((tree, tree));
 const struct attribute_spec ix86_attribute_table[];
+static bool ix86_function_ok_for_sibcall PARAMS ((tree, tree));
 static tree ix86_handle_cdecl_attribute PARAMS ((tree *, tree, tree, int, bool
*));
 static tree ix86_handle_regparm_attribute PARAMS ((tree *, tree, tree, int,
bool *));
 static int ix86_value_regno PARAMS ((enum machine_mode));
@@ -843,6 +844,9 @@ static enum x86_64_reg_class merge_class
 #define TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD \
   ia32_multipass_dfa_lookahead

+#undef TARGET_FUNCTION_OK_FOR_SIBCALL
+#define TARGET_FUNCTION_OK_FOR_SIBCALL ix86_function_ok_for_sibcall
+
 #ifdef HAVE_AS_TLS
 #undef TARGET_HAVE_TLS
 #define TARGET_HAVE_TLS true
@@ -1286,6 +1290,24 @@ const struct attribute_spec ix86_attribu
 #endif
   { NULL,        0, 0, false, false, false, NULL }
 };
+
+/* If PIC, we cannot make sibling calls to global functions
+   because the PLT requires %ebx live.
+   If we are returning floats on the register stack, we cannot make
+   sibling calls to functions that return floats.  (The stack adjust
+   instruction will wind up after the sibcall jump, and not be executed.)  */
+
+static bool
+ix86_function_ok_for_sibcall (decl, exp)
+     tree decl;
+     tree exp ATTRIBUTE_UNUSED;
+{
+  return ((decl)
+          && (! flag_pic || ! TREE_PUBLIC (decl))
+          && (! TARGET_FLOAT_RETURNS_IN_80387
+              || ! FLOAT_MODE_P (TYPE_MODE (TREE_TYPE (TREE_TYPE (decl))))
+              || FLOAT_MODE_P (TYPE_MODE (TREE_TYPE (TREE_TYPE
(cfun->decl))))));
+}

 /* Handle a "cdecl" or "stdcall" attribute;
    arguments as in struct attribute_spec.handler.  */
Index: config/i386/i386.h
===================================================================
RCS file: /cvsroot/gcc/gcc/gcc/config/i386/i386.h,v
retrieving revision 1.280.4.5
diff -u -p -r1.280.4.5 i386.h
--- config/i386/i386.h  23 Sep 2002 04:38:45 -0000      1.280.4.5
+++ config/i386/i386.h  1 Oct 2002 00:28:30 -0000
@@ -1674,18 +1674,6 @@ typedef struct ix86_args {
```

```
417
418    #define FUNCTION_ARG_PARTIAL_NREGS(CUM, MODE, TYPE, NAMED) 0
419
420   -/* If PIC, we cannot make sibling calls to global functions
421   -   because the PLT requires %ebx live.
422   -   If we are returning floats on the register stack, we cannot make
423   -   sibling calls to functions that return floats.  (The stack adjust
424   -   instruction will wind up after the sibcall jump, and not be executed.) */
425   -#define FUNCTION_OK_FOR_SIBCALL(DECL)                                      \
426   -  ((DECL)                                                                  \
427   -   && (! flag_pic || ! TREE_PUBLIC (DECL))                                 \
428   -   && (! TARGET_FLOAT_RETURNS_IN_80387                                     \
429   -       || ! FLOAT_MODE_P (TYPE_MODE (TREE_TYPE (TREE_TYPE (DECL))))    \
430   -       || FLOAT_MODE_P (TYPE_MODE (TREE_TYPE (TREE_TYPE (cfun->decl))))))
431   -
432    /* Perform any needed actions needed for a function that is receiving a
433       variable number of arguments.
434
435   Index: config/pa/pa-linux.h
436   ===================================================================
437   RCS file: /cvsroot/gcc/gcc/gcc/config/pa/pa-linux.h,v
438   retrieving revision 1.24.2.1
439   diff -u -p -r1.24.2.1 pa-linux.h
440   --- config/pa/pa-linux.h        2 Sep 2002 02:54:02 -0000       1.24.2.1
441   +++ config/pa/pa-linux.h        1 Oct 2002 00:28:33 -0000
442   @@ -81,10 +81,6 @@ Boston, MA 02111-1307, USA.  */
443          %{!dynamic-linker:-dynamic-linker /lib/ld.so.1}} \
444          %{static:-static}}"
445
446   -/* Sibcalls, stubs, and elf sections don't play well.  */
447   -#undef FUNCTION_OK_FOR_SIBCALL
448   -#define FUNCTION_OK_FOR_SIBCALL(x) 0
449   -
450    /* glibc's profiling functions don't need gcc to allocate counters.  */
451    #define NO_PROFILE_COUNTERS 1
452
453   @@ -172,6 +168,11 @@ Boston, MA 02111-1307, USA.  */
454    #undef TARGET_ASM_GLOBALIZE_LABEL
455    /* Globalizing directive for a label.  */
456    #define GLOBAL_ASM_OP ".globl "
457   +
458   +/* This definition is used inside pa.c to disable all
459   +   sibcall optimization, because sibcalls, stubs and
460   +   elf sections don't play well.  */
461   +#define TARGET_HAS_STUBS_AND_ELF_SECTIONS 1
462
463    /* FIXME: Hacked from the <elfos.h> one so that we avoid multiple
464       labels in a function declaration (since pa.c seems determined to do
465   Index: config/pa/pa.c
466   ===================================================================
467   RCS file: /cvsroot/gcc/gcc/gcc/config/pa/pa.c,v
468   retrieving revision 1.177.2.3
469   diff -u -p -r1.177.2.3 pa.c
470   --- config/pa/pa.c      17 Sep 2002 22:59:03 -0000      1.177.2.3
471   +++ config/pa/pa.c      1 Oct 2002 00:28:59 -0000
472   @@ -116,6 +116,7 @@ static void pa_select_section PARAMS ((t
473         ATTRIBUTE_UNUSED;
```

```
474     static void pa_encode_section_info PARAMS ((tree, int));
475     static const char *pa_strip_name_encoding PARAMS ((const char *));
476    +static bool pa_function_ok_for_sibcall PARAMS ((tree, tree));
477     static void pa_globalize_label PARAMS ((FILE *, const char *))
478          ATTRIBUTE_UNUSED;
479
480    @@ -194,6 +195,9 @@ static size_t n_deferred_plabels = 0;
481     #undef TARGET_STRIP_NAME_ENCODING
482     #define TARGET_STRIP_NAME_ENCODING pa_strip_name_encoding
483
484    +#undef TARGET_FUNCTION_OK_FOR_SIBCALL
485    +#define TARGET_FUNCTION_OK_FOR_SIBCALL pa_function_ok_for_sibcall
486    +
487     struct gcc_target targetm = TARGET_INITIALIZER;
488     ^L
489     void
490    @@ -6631,6 +6635,44 @@ pa_asm_output_mi_thunk (file, thunk_fnde
491           function_section (thunk_fndecl);
492         }
493       current_thunk_number++;
494    +}
495    +
496    +/* Only direct calls to static functions are allowed to be sibling (tail)
497    +   call optimized.
498    +
499    +   This restriction is necessary because some linker generated stubs will
500    +   store return pointers into rp' in some cases which might clobber a
501    +   live value already in rp'.
502    +
503    +   In a sibcall the current function and the target function share stack
504    +   space.  Thus if the path to the current function and the path to the
505    +   target function save a value in rp', they save the value into the
506    +   same stack slot, which has undesirable consequences.
507    +
508    +   Because of the deferred binding nature of shared libraries any function
509    +   with external scope could be in a different load module and thus require
510    +   rp' to be saved when calling that function.  So sibcall optimizations
511    +   can only be safe for static function.
512    +
513    +   Note that GCC never needs return value relocations, so we don't have to
514    +   worry about static calls with return value relocations (which require
515    +   saving rp').
516    +
517    +   It is safe to perform a sibcall optimization when the target function
518    +   will never return.  */
519    +static bool
520    +pa_function_ok_for_sibcall (decl, exp)
521    +     tree decl;
522    +     tree exp ATTRIBUTE_UNUSED;
523    +{
524    +#ifdef TARGET_HAS_STUBS_AND_ELF_SECTIONS
525    +  /* Sibcalls, stubs, and elf sections don't play well.  */
526    +  return false;
527    +#endif
528    +  return (decl
529    +          && ! TARGET_PORTABLE_RUNTIME
530    +          && ! TARGET_64BIT
```

```
531   +         && ! TREE_PUBLIC (decl));
532    }
533
534    /* Returns 1 if the 6 operands specified in OPERANDS are suitable for
535   Index: config/pa/pa.h
536   ===================================================================
537   RCS file: /cvsroot/gcc/gcc/gcc/config/pa/pa.h,v
538   retrieving revision 1.166.2.4
539   diff -u -p -r1.166.2.4 pa.h
540   --- config/pa/pa.h      17 Sep 2002 22:59:03 -0000      1.166.2.4
541   +++ config/pa/pa.h      1 Oct 2002 00:29:07 -0000
542   @@ -1831,35 +1831,6 @@ do {
543   \
544    /* The number of Pmode words for the setjmp buffer.  */
545    #define JMP_BUF_SIZE 50
546
547   -/* Only direct calls to static functions are allowed to be sibling (tail)
548   -   call optimized.
549   -
550   -   This restriction is necessary because some linker generated stubs will
551   -   store return pointers into rp' in some cases which might clobber a
552   -   live value already in rp'.
553   -
554   -   In a sibcall the current function and the target function share stack
555   -   space.  Thus if the path to the current function and the path to the
556   -   target function save a value in rp', they save the value into the
557   -   same stack slot, which has undesirable consequences.
558   -
559   -   Because of the deferred binding nature of shared libraries any function
560   -   with external scope could be in a different load module and thus require
561   -   rp' to be saved when calling that function.  So sibcall optimizations
562   -   can only be safe for static function.
563   -
564   -   Note that GCC never needs return value relocations, so we don't have to
565   -   worry about static calls with return value relocations (which require
566   -   saving rp').
567   -
568   -   It is safe to perform a sibcall optimization when the target function
569   -   will never return.  */
570   -#define FUNCTION_OK_FOR_SIBCALL(DECL) \
571   -  (DECL \
572   -   && ! TARGET_PORTABLE_RUNTIME \
573   -   && ! TARGET_64BIT \
574   -   && ! TREE_PUBLIC (DECL))
575   -
576    #define PREDICATE_CODES
577   \
578      {"reg_or_0_operand", {SUBREG, REG, CONST_INT}},                      \
579      {"call_operand_address", {LABEL_REF, SYMBOL_REF, CONST_INT,          \
580   Index: config/rs6000/rs6000-protos.h
581   ===================================================================
582   RCS file: /cvsroot/gcc/gcc/gcc/config/rs6000/rs6000-protos.h,v
583   retrieving revision 1.43.4.1
584   diff -u -p -r1.43.4.1 rs6000-protos.h
585   --- config/rs6000/rs6000-protos.h       17 Sep 2002 22:59:05 -0000      1.43.4.1
586   +++ config/rs6000/rs6000-protos.h       1 Oct 2002 00:29:09 -0000
587   @@ -151,7 +151,6 @@ extern void setup_incoming_varargs PARAM
```

```
588                                             int *, int));
589  extern struct rtx_def *rs6000_va_arg PARAMS ((tree, tree));
590  extern void output_mi_thunk PARAMS ((FILE *, tree, int, tree));
591 -extern int function_ok_for_sibcall PARAMS ((tree));
592  #ifdef ARGS_SIZE_RTX
593  /* expr.h defines ARGS_SIZE_RTX and `enum direction' */
594  extern enum direction function_arg_padding PARAMS ((enum machine_mode, tree));
595 Index: config/rs6000/rs6000.c
596 ===================================================================
597 RCS file: /cvsroot/gcc/gcc/gcc/config/rs6000/rs6000.c,v
598 retrieving revision 1.366.2.5
599 diff -u -p -r1.366.2.5 rs6000.c
600 --- config/rs6000/rs6000.c      20 Sep 2002 01:29:19 -0000      1.366.2.5
601 +++ config/rs6000/rs6000.c      1 Oct 2002 00:29:54 -0000
602 @@ -165,6 +165,7 @@ struct builtin_description
603    const enum rs6000_builtins code;
604  };
605
606 +static bool rs6000_function_ok_for_sibcall PARAMS ((tree, tree));
607  static void rs6000_add_gc_roots PARAMS ((void));
608  static int num_insns_constant_wide PARAMS ((HOST_WIDE_INT));
609  static void validate_condition_mode
610 @@ -376,6 +377,9 @@ static const char alt_reg_names[][8] =
611  /* The VRSAVE bitmask puts bit %v0 as the most significant bit.  */
612  #define ALTIVEC_REG_BIT(REGNO) (0x80000000 >> ((REGNO) - FIRST_ALTIVEC_REGNO))
613
614 +#undef TARGET_FUNCTION_OK_FOR_SIBCALL
615 +#define TARGET_FUNCTION_OK_FOR_SIBCALL rs6000_function_ok_for_sibcall
616 +
617  struct gcc_target targetm = TARGET_INITIALIZER;
618  ^L
619  /* Override command line options.  Mostly we process the processor
620 @@ -9403,33 +9407,34 @@ rs6000_return_addr (count, frame)
621     vector parameters are required to have a prototype, so the argument
622     type info must be available here.  (The tail recursion case can work
623     with vector parameters, but there's no way to distinguish here.) */
624 -int
625 -function_ok_for_sibcall (fndecl)
626 -    tree fndecl;
627 +static bool
628 +rs6000_function_ok_for_sibcall (decl, exp)
629 +    tree decl;
630 +    tree exp ATTRIBUTE_UNUSED;
631  {
632    tree type;
633 -  if (fndecl)
634 +  if (decl)
635      {
636        if (TARGET_ALTIVEC_VRSAVE)
637          {
638 -         for (type = TYPE_ARG_TYPES (TREE_TYPE (fndecl));
639 +         for (type = TYPE_ARG_TYPES (TREE_TYPE (decl));
640                type; type = TREE_CHAIN (type))
641            {
642              if (TREE_CODE (TREE_VALUE (type)) == VECTOR_TYPE)
643 -              return 0;
644 +              return false;
```

```
645               }
646             }
647           if (DEFAULT_ABI == ABI_DARWIN
648   -         || (*targetm.binds_local_p) (fndecl))
649   +         || (*targetm.binds_local_p) (decl))
650           {
651   -         tree attr_list = TYPE_ATTRIBUTES (TREE_TYPE (fndecl));
652   +         tree attr_list = TYPE_ATTRIBUTES (TREE_TYPE (decl));
653
654             if (!lookup_attribute ("longcall", attr_list)
655                 || lookup_attribute ("shortcall", attr_list))
656   -           return 1;
657   +           return true;
658           }
659       }
660   -  return 0;
661   +  return false;
662    }
663
664    /* function rewritten to handle sibcalls */
665   Index: config/rs6000/rs6000.h
666   ===================================================================
667   RCS file: /cvsroot/gcc/gcc/gcc/config/rs6000/rs6000.h,v
668   retrieving revision 1.224.4.6
669   diff -u -p -r1.224.4.6 rs6000.h
670   --- config/rs6000/rs6000.h      23 Sep 2002 04:38:47 -0000      1.224.4.6
671   +++ config/rs6000/rs6000.h      1 Oct 2002 00:30:07 -0000
672   @@ -1804,10 +1804,6 @@ typedef struct rs6000_args
673       argument is passed depends on whether or not it is a named argument.  */
674    #define STRICT_ARGUMENT_NAMING 1
675
676   -/* We do not allow indirect calls to be optimized into sibling calls, nor
677   -   do we allow calls with vector parameters.  */
678   -#define FUNCTION_OK_FOR_SIBCALL(DECL) function_ok_for_sibcall ((DECL))
679   -
680    /* Output assembler code to FILE to increment profiler label # LABELNO
681       for profiling a function entry.  */
682
683   Index: config/sh/sh.c
684   ===================================================================
685   RCS file: /cvsroot/gcc/gcc/gcc/config/sh/sh.c,v
686   retrieving revision 1.169.4.4
687   diff -u -p -r1.169.4.4 sh.c
688   --- config/sh/sh.c      20 Sep 2002 01:29:21 -0000      1.169.4.4
689   +++ config/sh/sh.c      1 Oct 2002 00:30:32 -0000
690   @@ -199,6 +199,7 @@ static void sh_insert_attributes PARAMS
691    static int sh_adjust_cost PARAMS ((rtx, rtx, rtx, int));
692    static int sh_use_dfa_interface PARAMS ((void));
693    static int sh_issue_rate PARAMS ((void));
694   +static bool sh_function_ok_for_sibcall PARAMS ((tree, tree));
695
696    static bool sh_cannot_modify_jumps_p PARAMS ((void));
697    static bool sh_ms_bitfield_layout_p PARAMS ((tree));
698   @@ -259,6 +260,9 @@ static void flow_dependent_p_1 PARAMS ((
699    #undef TARGET_EXPAND_BUILTIN
700    #define TARGET_EXPAND_BUILTIN sh_expand_builtin
701
```

```
+#undef TARGET_FUNCTION_OK_FOR_SIBCALL
+#define TARGET_FUNCTION_OK_FOR_SIBCALL sh_function_ok_for_sibcall
+
 struct gcc_target targetm = TARGET_INITIALIZER;
 ^L
 /* Print the operand address in x to the stream.  */
@@ -7383,6 +7387,19 @@ sh_initialize_trampoline (tramp, fnaddr,
     }
 }

+/* FIXME: This is overly conservative.  A SHcompact function that
+   receives arguments ``by reference'' will have them stored in its
+   own stack frame, so it must not pass pointers or references to
+   these arguments to other functions by means of sibling calls.  */
+static bool
+sh_function_ok_for_sibcall (decl, exp)
+     tree decl;
+     tree exp ATTRIBUTE_UNUSED;
+{
+  return (decl
+         && (! TARGET_SHCOMPACT
+             || current_function_args_info.stack_regs == 0));
+}
 ^L
 /* Machine specific built-in functions.  */

Index: config/sh/sh.h
===================================================================
RCS file: /cvsroot/gcc/gcc/gcc/config/sh/sh.h,v
retrieving revision 1.166.4.6
diff -u -p -r1.166.4.6 sh.h
--- config/sh/sh.h      23 Sep 2002 04:38:48 -0000      1.166.4.6
+++ config/sh/sh.h      1 Oct 2002 00:30:46 -0000
@@ -1706,13 +1706,6 @@ struct sh_args {
     (CUM).outgoing = 0;                                                \
   } while (0)

-/* FIXME: This is overly conservative.  A SHcompact function that
-   receives arguments ``by reference'' will have them stored in its
-   own stack frame, so it must not pass pointers or references to
-   these arguments to other functions by means of sibling calls.  */
-#define FUNCTION_OK_FOR_SIBCALL(DECL) \
-  (! TARGET_SHCOMPACT || current_function_args_info.stack_regs == 0)
-
 /* Update the data in CUM to advance over an argument
    of mode MODE and data type TYPE.
    (TYPE is null for libcalls where that information may not be
Index: config/sparc/sparc.c
===================================================================
RCS file: /cvsroot/gcc/gcc/gcc/config/sparc/sparc.c,v
retrieving revision 1.226.4.4
diff -u -p -r1.226.4.4 sparc.c
--- config/sparc/sparc.c        20 Sep 2002 01:29:21 -0000      1.226.4.4
+++ config/sparc/sparc.c        1 Oct 2002 00:31:13 -0000
@@ -176,6 +176,8 @@ static void emit_soft_tfmode_cvt PARAMS
 static void emit_hard_tfmode_operation PARAMS ((enum rtx_code, rtx *));

```

```
759    static void sparc_encode_section_info PARAMS ((tree, int));
760    +
761    +static bool sparc_function_ok_for_sibcall PARAMS ((tree, tree));
762     ^L
763     /* Option handling.  */
764
765    @@ -239,6 +241,9 @@ enum processor_type sparc_cpu;
766     #undef TARGET_ENCODE_SECTION_INFO
767     #define TARGET_ENCODE_SECTION_INFO sparc_encode_section_info
768
769    +#undef TARGET_FUNCTION_OK_FOR_SIBCALL
770    +#define TARGET_FUNCTION_OK_FOR_SIBCALL sparc_function_ok_for_sibcall
771    +
772     struct gcc_target targetm = TARGET_INITIALIZER;
773     ^L
774     /* Validate and override various options, and do some machine dependent
775    @@ -8021,6 +8026,32 @@ sparc_elf_asm_named_section (name, flags
776        fputc ('\n', asm_out_file);
777     }
778     #endif /* OBJECT_FORMAT_ELF */
779    +
780    +/* We do not allow sibling calls if -mflat, nor
781    +   we do not allow indirect calls to be optimized into sibling calls.
782    +
783    +   Also, on sparc 32-bit we cannot emit a sibling call when the
784    +   current function returns a structure.  This is because the "unimp
785    +   after call" convention would cause the callee to return to the
786    +   wrong place.  The generic code already disallows cases where the
787    +   function being called returns a structure.
788    +
789    +   It may seem strange how this last case could occur.  Usually there
790    +   is code after the call which jumps to epilogue code which dumps the
791    +   return value into the struct return area.  That ought to invalidate
792    +   the sibling call right?  Well, in the c++ case we can end up passing
793    +   the pointer to the struct return area to a constructor (which returns
794    +   void) and then nothing else happens.  Such a sibling call would look
795    +   valid without the added check here.  */
796    +static bool
797    +sparc_function_ok_for_sibcall (decl, exp)
798    +     tree decl;
799    +     tree exp ATTRIBUTE_UNUSED;
800    +{
801    +  return (decl
802    +          && ! TARGET_FLAT
803    +          && (TARGET_ARCH64 || ! current_function_returns_struct));
804    +}
805
806     /* ??? Similar to the standard section selection, but force reloc-y-ness
807        if SUNOS4_SHARED_LIBRARIES.  Unclear why this helps (as opposed to
808    Index: config/sparc/sparc.h
809    ===================================================================
810    RCS file: /cvsroot/gcc/gcc/gcc/config/sparc/sparc.h,v
811    retrieving revision 1.207.4.6
812    diff -u -p -r1.207.4.6 sparc.h
813    --- config/sparc/sparc.h        23 Sep 2002 04:38:48 -0000      1.207.4.6
814    +++ config/sparc/sparc.h        1 Oct 2002 00:31:25 -0000
815    @@ -1934,27 +1934,6 @@ do {
```

```
816   \
817
818    #define STRICT_ARGUMENT_NAMING TARGET_V9
819
820   -/* We do not allow sibling calls if -mflat, nor
821   -   we do not allow indirect calls to be optimized into sibling calls.
822   -
823   -   Also, on sparc 32-bit we cannot emit a sibling call when the
824   -   current function returns a structure.  This is because the "unimp
825   -   after call" convention would cause the callee to return to the
826   -   wrong place.  The generic code already disallows cases where the
827   -   function being called returns a structure.
828   -
829   -   It may seem strange how this last case could occur.  Usually there
830   -   is code after the call which jumps to epilogue code which dumps the
831   -   return value into the struct return area.  That ought to invalidate
832   -   the sibling call right?  Well, in the c++ case we can end up passing
833   -   the pointer to the struct return area to a constructor (which returns
834   -   void) and then nothing else happens.  Such a sibling call would look
835   -   valid without the added check here.  */
836   -#define FUNCTION_OK_FOR_SIBCALL(DECL) \
837   -       (DECL \
838   -        && ! TARGET_FLAT \
839   -        && (TARGET_ARCH64 || ! current_function_returns_struct))
840   -
841    /* Generate RTL to flush the register windows so as to make arbitrary frames
842       available.  */
843    #define SETUP_FRAME_ADDRESSES()                \
844   Index: config/xtensa/xtensa.h
845   ===================================================================
846   RCS file: /cvsroot/gcc/gcc/gcc/config/xtensa/xtensa.h,v
847   retrieving revision 1.20.4.1
848   diff -u -p -r1.20.4.1 xtensa.h
849   --- config/xtensa/xtensa.h      16 Sep 2002 17:38:28 -0000      1.20.4.1
850   +++ config/xtensa/xtensa.h      1 Oct 2002 00:31:41 -0000
851   @@ -1287,11 +1287,6 @@ typedef struct xtensa_args {
852       indexing purposes) so give the MEM rtx a words's mode.  */
853    #define FUNCTION_MODE SImode
854
855   -/* A C expression that evaluates to true if it is ok to perform a
856   -   sibling call to DECL.  */
857   -/* TODO: fix this up to allow at least some sibcalls */
858   -#define FUNCTION_OK_FOR_SIBCALL(DECL) 0
859   -
860    /* Xtensa constant costs.  */
861    #define CONST_COSTS(X, CODE, OUTER_CODE)                              \
862      case CONST_INT:                                                     \
```

Patch 2: Documentation Updates

```
01   2002-10-03  Andreas Bauer  <baueran@in.tum.de>
02
03           doc/tm.texi (FUNCTION_OK_FOR_SIBCALL): Remove.
04           (TARGET_FUNCTION_OK_FOR_SIBCALL): New.
05
06   Index: tm.texi
07   ===================================================================
```

```
RCS file: /cvsroot/gcc/gcc/gcc/doc/tm.texi,v
retrieving revision 1.159.2.7
diff -u -p -r1.159.2.7 tm.texi
--- tm.texi     1 Oct 2002 17:32:35 -0000       1.159.2.7
+++ tm.texi     3 Oct 2002 06:46:26 -0000
@@ -4228,18 +4228,18 @@ the function prologue.  Normally, the pr
 @subsection Permitting tail calls
 @cindex tail calls

-@table @code
-@findex FUNCTION_OK_FOR_SIBCALL
-@item FUNCTION_OK_FOR_SIBCALL (@var{decl})
-A C expression that evaluates to true if it is ok to perform a sibling
-call to @var{decl} from the current function.
+@deftypefn {Target Hook} bool TARGET_FUNCTION_OK_FOR_SIBCALL (tree @var{decl},
tree @var{exp})
+True if it is ok to do sibling call optimization for the specified
+call expression @var{exp}.  @var{decl} will be the called function,
+or NULL if this is an indirect call.

 It is not uncommon for limitations of calling conventions to prevent
 tail calls to functions outside the current unit of translation, or
-during PIC compilation.  Use this macro to enforce these restrictions,
+during PIC compilation.  The hook is used to enforce these restrictions,
 as the @code{sibcall} md pattern can not fail, or fall over to a
-``normal'' call.
-@end table
+``normal'' call.  The criteria for successful sibling call optimization
+may vary greatly between different architectures.
+@end deftypefn

 @node Varargs
 @section Implementing the Varargs Macros
```

Patch 3: Bug Fixes

```
2002-10-07  Andreas Bauer  <baueran@in.tum.de>

        * calls.c (expand_call): Fix function-is-volatile
        check.

Index: calls.c
===================================================================
RCS file: /cvsroot/gcc/gcc/gcc/calls.c,v
retrieving revision 1.231.4.7
diff -u -p -r1.231.4.7 calls.c
--- calls.c     1 Oct 2002 20:22:34 -0000       1.231.4.7
+++ calls.c     7 Oct 2002 03:49:59 -0000
@@ -2444,7 +2444,7 @@ expand_call (exp, target, ignore)
        || !(*targetm.function_ok_for_sibcall) (fndecl, exp)
        || (flags & (ECF_RETURNS_TWICE | ECF_LONGJMP))
        /* Functions that do not return may not be sibcall optimized.  */
-       || TYPE_VOLATILE (TREE_TYPE (TREE_OPERAND (exp, 0)))
+       || TYPE_VOLATILE (TREE_TYPE (TREE_TYPE (TREE_OPERAND (exp, 0))))
        /* If this function requires more stack slots than the current
          function, we cannot change it into a sibling call.  */
        || args_size.constant > current_function_args_size
```

Patch 4: Modifying the Call Patterns

```
2002-10-08  Andreas Bauer  <baueran@in.tum.de>

        * config/i386/i386.c (ix86_function_ok_for_sibcall): Allow
        indirect calls to be sibcall optimized.
        * config/i386/i386.md (sibcall_1): New.
        (call_1): Add no-sibcalls condition.
        (sibcall_value_1): New.
        (call_value_1): Add no-sibcalls condition.

Index: i386.c
===================================================================
RCS file: /cvsroot/gcc/gcc/gcc/config/i386/i386.c,v
retrieving revision 1.447.2.6
diff -u -p -r1.447.2.6 i386.c
--- i386.c      5 Oct 2002 21:27:49 -0000       1.447.2.6
+++ i386.c      8 Oct 2002 01:39:04 -0000
@@ -1305,13 +1305,28 @@ const struct attribute_spec ix86_attribu
 static bool
 ix86_function_ok_for_sibcall (decl, exp)
      tree decl;
-     tree exp ATTRIBUTE_UNUSED;
+     tree exp;
 {
-  return ((decl)
-          && (! flag_pic || ! TREE_PUBLIC (decl))
-          && (! TARGET_FLOAT_RETURNS_IN_80387
-              || ! FLOAT_MODE_P (TYPE_MODE (TREE_TYPE (TREE_TYPE (decl))))
-              || FLOAT_MODE_P (TYPE_MODE (TREE_TYPE (TREE_TYPE
(cfun->decl))))));
+  /* If we are generating position-independent code, we cannot sibcall
+     optimize any indirect call, or a direct call to a global function,
+     as the PLT requires %ebx be live.  */
+  if (flag_pic && (!decl || !TREE_PUBLIC (decl)))
+    return 0;
+
+  /* If we are returning floats on the 80387 register stack, we cannot
+     make a sibcall from a function that doesn't return a float to a
+     function that does; the necessary stack adjustment will not be
+     executed.  */
+  if (TARGET_FLOAT_RETURNS_IN_80387
+      && FLOAT_MODE_P (TYPE_MODE (TREE_TYPE (exp)))
+      && !FLOAT_MODE_P (TYPE_MODE (TREE_TYPE (TREE_TYPE (cfun->decl)))))
+    return 0;
+
+  /* Otherwise okay.
+     That also includes certain types of indirect calls, since DECL
+     is not necessarily defined here and the i386 machine description
+     supports the according call patterns.  */
+  return 1;
 }

 /* Handle a "cdecl" or "stdcall" attribute;
Index: i386.md
===================================================================
RCS file: /cvsroot/gcc/gcc/gcc/config/i386/i386.md,v
retrieving revision 1.380.4.5
```

```
diff -u -p -r1.380.4.5 i386.md
--- i386.md     1 Oct 2002 17:32:07 -0000       1.380.4.5
+++ i386.md     8 Oct 2002 01:39:07 -0000
@@ -13427,19 +13427,22 @@
 (define_insn "*call_1"
   [(call (mem:QI (match_operand:SI 0 "call_insn_operand" "rsm"))
         (match_operand 1 "" ""))]
-  "!TARGET_64BIT"
+  "!SIBLING_CALL_P (insn) && !TARGET_64BIT"
 {
   if (constant_call_address_operand (operands[0], QImode))
-    {
-      if (SIBLING_CALL_P (insn))
-       return "jmp\t%P0";
-      else
-       return "call\t%P0";
-    }
-  if (SIBLING_CALL_P (insn))
-    return "jmp\t%A0";
-  else
-    return "call\t%A0";
+    return "call\t%P0";
+  return "call\t%A0";
+}
+  [(set_attr "type" "call")])
+
+(define_insn "*sibcall_1"
+  [(call (mem:QI (match_operand:SI 0 "call_insn_operand" "s,c,d,a"))
+        (match_operand 1 "" ""))]
+  "SIBLING_CALL_P (insn) && !TARGET_64BIT"
+{
+  if (constant_call_address_operand (operands[0], QImode))
+    return "jmp\t%P0";
+  return "jmp\t%A0";
 }
   [(set_attr "type" "call")])

@@ -17716,19 +17719,23 @@
   [(set (match_operand 0 "" "")
        (call (mem:QI (match_operand:SI 1 "call_insn_operand" "rsm"))
              (match_operand:SI 2 "" "")))]
-  "!TARGET_64BIT"
+  "!SIBLING_CALL_P (insn) && !TARGET_64BIT"
 {
   if (constant_call_address_operand (operands[1], QImode))
-    {
-      if (SIBLING_CALL_P (insn))
-       return "jmp\t%P1";
-      else
-       return "call\t%P1";
-    }
-  if (SIBLING_CALL_P (insn))
-    return "jmp\t%*%1";
-  else
-    return "call\t%*%1";
+    return "call\t%P1";
+  return "call\t%*%1";
```

```
114  +}
115  +  [(set_attr "type" "callv")])
116  +
117  +(define_insn "*sibcall_value_1"
118  +  [(set (match_operand 0 "" "")
119  +        (call (mem:QI (match_operand:SI 1 "call_insn_operand" "s,c,d,a"))
120  +              (match_operand:SI 2 "" ""))))]
121  +  "SIBLING_CALL_P (insn) && !TARGET_64BIT"
122  +{
123  +  if (constant_call_address_operand (operands[1], QImode))
124  +    return "jmp\t%P1";
125  +  return "jmp\t%*%1";
126   }
127      [(set_attr "type" "callv")])
```

Patch 5: Fix Typo in Documentation

```
01   2002-10-28  Andreas Bauer  <baueran@in.tum.de>
02
03           doc/c-tree.texi (Tree overview): Fix typos.
04
05   Index: c-tree.texi
06   ===================================================================
07   RCS file: /cvsroot/gcc/gcc/gcc/doc/c-tree.texi,v
08   retrieving revision 1.33.4.2
09   diff -u -w -r1.33.4.2 c-tree.texi
10   --- c-tree.texi 21 Oct 2002 17:52:58 -0000      1.33.4.2
11   +++ c-tree.texi 28 Oct 2002 00:52:41 -0000
12   @@ -84,8 +84,8 @@
13    font}, except when talking about the actual C type @code{tree}.
14
15    You can tell what kind of node a particular tree is by using the
16   -@code{TREE_CODE} macro.  Many, many macros take a trees as input and
17   -return trees as output.  However, most macros require a certain kinds of
18   +@code{TREE_CODE} macro.  Many, many macros take trees as input and
19   +return trees as output.  However, most macros require a certain kind of
20    tree node as input.  In other words, there is a type-system for trees,
21    but it is not reflected in the C type-system.
```

Patch 6: Enhance GCC Test Suite

```
01   2002-11-04  Andreas Bauer  <baueran@in.tum.de>
02
03           * gcc.dg/sibcall-6: New test for indirect sibcalls.
04
05   --- /dev/null   Sun Jul 14 11:06:13 2002
06   +++ sibcall-6.c Tue Nov  5 11:03:03 2002
07   @@ -0,0 +1,42 @@
08   +/* A simple check to see whether indirect calls are
09   +   being sibcall optimized on targets that do support
10   +   this notion, i.e. have the according call patterns
11   +   in place.
12   +
13   +   Copyright (C) 2002 Free Software Foundation Inc.
14   +   Contributed by Andreas Bauer <baueran@in.tum.de>  */
15   +
16   +/* { dg-do run { target i?86-*-* x86_64-*-*} } */
```

```
17   +/* { dg-options "-O2 -foptimize-sibling-calls" } */
18   +
19   +int foo (int);
20   +int bar (int);
21   +
22   +int (*ptr) (int);
23   +int *f_addr;
24   +
25   +int
26   +main ()
27   +{
28   +  ptr = bar;
29   +  foo (7);
30   +  exit (0);
31   +}
32   +
33   +int
34   +bar (b)
35   +     int b;
36   +{
37   +  if (f_addr == (int*) __builtin_return_address (0))
38   +    return b;
39   +  else
40   +    abort ();
41   +}
42   +
43   +int
44   +foo (f)
45   +     int f;
46   +{
47   +  f_addr = (int*) __builtin_return_address (0);
48   +  return (*ptr)(f);
49   +}
```

C.2 SUPER SIBCALLS

The patch for Super Sibcalls exceeds 2000 lines of code, which is the reason why it is not fully printed in this document. Instead, the changes are available from the attached CD-ROM, or can be obtained by sending an e-mail to the author and, alternatively, by visiting his homepage at `http://www.andreasbauer.org/`.

Due to the fact that, the patch is not part of mainline GCC, it has been made against the *stable* version 3.2 of GCC. This way, others can do some testing without having to download a specific CVS version of the compiler suite. In its current state, however, the patch is not fully functional and, therefore, should not be applied without accompanying modifications.

C.3 AN APPLICATION: NEWTON SQUARE ROOT

"Newton Square Root" is a perfect example for a very useful and also tail recursive algorithm [Knuth 1998b, § 4.3.1, 4.3.3]. However, the following implementation is rather artificial in a sense that the tail recursive calls have

been transformed into mutually recursive calls, because some versions of GCC would be able to transform "ordinary" recursion into `goto`'s, if a sufficiently high level of optimisation is enabled. According to § 3.2, the goto optimisation is not handled by the sibcall mechanism and would, therefore, not be suited to demonstrate the impact of tail calls (or the lack thereof) on common C programs.

Due to the fact that the functions are mostly free from local variable declarations, the mutual recursion has been turned into indirect calls, because indirect sibling calls have not been possible with *any* GCC version prior to 3.4 (which contains this thesis' extensions). Direct sibling calls, on the other hand, were previously featured in GCC, if the stringent criteria described in § 3.3 hold.

So if this code is compiled on GCC 3.4 (or newer), it will not abort in a similar fashion as it is described in § 1.2, because the calls in the tail position can be transformed into jump commands which do not reserve space for additional stack frames.

```
01  /* Newton's Square Root algorithm,     *
02   * Implemented in C by Andreas Bauer.  */
03
04  #include <stdio.h>
05  #include <stdlib.h>
06  #include <math.h>
07
08  #define ACCURACY 0.0000001
09
10  float newton (int);
11  float find_root (float, float);
12  float find_root2 (float, float);
13  int close_enough (float, float);
14
15  int calls;
16  float (*find_root_ptr) (float, float);
17  float (*find_root_ptr2) (float, float);
18
19  int main (void)
20  {
21    int input = 0;
22    find_root_ptr = find_root;
23    find_root_ptr2 = find_root2;
24    calls = 1;
25
26    printf ("Enter a number: ");
27    scanf ("%d", &input);
28    printf ("The square root of %d is approx. %f.\n", input, newton (input));
29    printf ("Function calls required: %d\n", calls);
30
31    return 0;
32  }
33
34  float newton (int input)
35  {
36    calls++;
37    return find_root ((float) input, 1);
38  }
```

```
39
40  float find_root (float input, float guess)
41  {
42    if (close_enough (input / guess, guess))
43      return guess;
44    else
45      {
46        calls++;
47
48        /* Indirect tail call.  */
49        return (*find_root_ptr2) (input, (guess + input / guess) / 2);
50      }
51  }
52
53  float find_root2 (float input, float guess)
54  {
55    if (close_enough (input / guess, guess))
56      return guess;
57    else
58      {
59        calls++;
60
61        /* Indirect tail call.  */
62        return (*find_root_ptr) (input, (guess + input / guess) / 2);
63      }
64  }
65
66  int close_enough (float a, float b)
67  {
68    return (fabs (a - b) < ACCURACY);
69  }
```

APPENDIX D

HARDWARE AND SOFTWARE USED

The main work for this thesis was undertaken using the UNIX-like operating system GNU/Linux version 2.4.7 on an Intel-related hardware platform whose components correspond to the following properties: AMD Duron processor, 1 GHz (2,000 bogomips), and 228 MBytes RAM. (The correct GCC target/host description for the deployed system is `i686-pc-linux-gnu`.) This resembles a fairly standard system which many people use at home, or in their office. Descriptions of the deployed graphic card, hard disks, the screen, and other peripherals are irrelevant for this work.[1]

The GCC source code was modified using the Emacs display editor (see Appendix B), version 21.2.1; errors were (sometimes) detected with the GNU Debugger (see Appendix B), and patches, make files and similar objects created by using the standard UNIX text tools and filters as they are explained, for example, in *The UNIX Programming Environment* [Kernighan and Pike 1984].

Testing GCC code changes for foreign platforms (ARM, PowerPC, etc.) was undertaken solely on the above system by using many different cross compilers which can be installed in parallel without affecting each other (see § 2.3).

[1] However, it should be pointed out that this work could not have been finished without a "sufficiently fast" Internet connection, because a GCC CVS branch is approximately 165 MBytes in size and changes to it have to be downloaded daily.

BIBLIOGRAPHY

Aho, A., Sethi, R. and Ullman, J. (1986). *Compilers — Principles, Techniques, and Tools*. Addison Wesley Higher Education.

American National Standard for Information Systems (1989). *Programming Language — C, ANSI X3.159-1989*. American National Standard for Information Systems (ANSI), New York.

ARM (2000). *The ARM-THUMB Procedure Call Standard*. SWS ESPC 0002 B-01, ARM Limited (Development Systems Business Unit, Engineering Software Group).

Baker, H. G. (1995). CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. In *ACM Sigplan Notices 30 (9)*. Association for Computing Machinery (ACM), New York.

Bartlett, J. F. (1989). SCHEME->C: a Portable Scheme-to-C Compiler. WRL Technical Report 89/1, DEC Western Research Laboratory, Palo Alto, California.

Brey, R. B. (1995). *The Intel 32-Bit Microprocessors/80386, 80486, and Pentium*. Prentice Hall New Jersey, Columbus, Ohio.

Chassell, R. J. and Stallman, R. M. (1999). *Texinfo: The GNU Documentation Format*. 4th ed. Free Software Foundation, Inc./GNU Press, Cambridge, Massachusetts.

Clinger, W. D. (1998). Proper tail recursion and space efficiency. *Proceedings of the 2002 ACM International Conference on Functional Programming*.

Conway, T., Henderson, F. and Somogyi, Z. (1995). Code generation for Mercury. In *Proceedings of the 1995 International Symposium on Logic Programming*. Portland, Oregon.

Cubranic, D. (1999). Open-Source Software Development. In *Proceedings of the ICSE-99 Workshop on Software Engineering over the Internet*. Los Angeles, California.

Free Software Foundation (1991). GNU General Public License. `http://www.fsf.org/licenses/gpl.html`, Free Software Foundation, Inc., Cambridge, Massachusetts.

Gudeman, D., De Bosschere, K. and Debray, S. K. (1992). An Efficient and Portable Sequential Implementation of Janus. In *Proceedings of the Joint International Conference and Symposium on Logic Programming* (Apt, ed.). The MIT Press, Washington.

Gunnarsson, H., Lundqvist, T. and Ernbert, B. (1995). Porting the GNU C Compiler to the Thor Microprocessor. Master Thesis Project, Saab Ericsson Space AB.

Henderson, F., Conway, T. and Somogyi, Z. (1995). Compiling logic programs to C using GNU C as a portable assembler. In *Proceedings of the ILPS '95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming Languages*. Portland, Oregon.

Hubicka, J., Jaeger, A. and Mitchell, M. (2002). *System V Application Binary Interface/x86-64 Architecture Processor Supplement (Draft Version 0.21)*. `http://www.x86-64.org/`.

IEEE Computer Society (1991). *IEEE Standard for the Scheme Programming Language*. IEEE standard 1178-1990 ed. The Institute of Electrical and Electronics Engineers, Inc., New York.

Intel Corporation (2001). *Intel Itanium Software Conventions and Runtime Architecture Guide*. Intel document SC–2791, Rev. No. 2.4E ed. Intel Corporation, Santa Clara, California.

ISO/IEC JTC1/SC22/WG14 (1999). Rationale for International Standard Programming Languages. Draft C99 Rationale, WG14 (C Standards Committee), Santa Cruz, California.

Kernighan, B. and Pike, R. (1984). *The UNIX Programming Environment*. Prentice Hall, New Jersey.

Kernighan, B. and Ritchie, D. (1988). *The C Programming Language*. 2nd ed. Prentice Hall, New Jersey.

Knuth, D. E. (1998a). *The Art of Computer Programming, Volume 1/Fundamental Algorithms*. 3rd ed. Addison Wesley, Boston.

Knuth, D. E. (1998b). *The Art of Computer Programming, Volume 2/Seminumerical Algorithms*. 3rd ed. Addison Wesley, Boston.

Knuth, D. E. (1999). *MMIXware: A RISC Computer for the Third Millennium*. Springer-Verlag, Heidelberg.

Nenzén, P. and Rågård, A. (2000). Tail Call Elimination in GCC. Bachelor's Project, Karlstad University, Sweden.

Nilsson, H.-P. (2001). GCC for MMIX: The ABI. `http://bitrange.com/mmix/mmixfest-2001/mmixabi.html`.

Peyton Jones, S., Hall, C., Hammond, K., Partain, W. and Wadler, P. (1992). The Glasgow Haskell compiler: a technical overview. *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference, Keele, 1993.*

Peyton Jones, S., Ramsey, N. and Reig, F. (1999). `C--`: a Portable Assembly Language that Supports Garbage Collection. In *International Conference on Principles and Practice of Declarative Programming.*

Pizka, M. (1997). Design and Implementation of the GNU INSEL Compiler gic. Technical Report TUM–I 9713, Munich University of Technology, Munich, Germany.

Pizka, M. (2002). The Portable Assembly Language `C--`: A Critical Review and a GCC Based Prototype. Internal Notes, Microsoft Research, Cambridge, UK.

Probst, M. (2001). *Proper Tail Recursion in C.* Master's thesis, Institut für Computersprachen der Technischen Universität Wien.

Salus, P. H. (1994). *A Quarter Century of UNIX.* Addison Wesley, Boston.

Serrano, M. and Weis, P. (1995). Bigloo: a portable and optimizing compiler for strict functional languages. In *Proceedings of the 2nd Symposium on Static Analysis.* Glasgow, Scotland.

Stallman, R. M. (2002). GNU Compiler Collection Internals. `http://gcc.gnu.org/onlinedocs/gccint/`, Free Software Foundation, Inc., Cambridge, Massachusetts.

The Santa Cruz Operation (1996). *System V Application Binary Interface/Intel386 Architecture Processor Supplement.* 4th ed. The Santa Cruz Operation, Inc. (SCO).

Vaughan, G. V., Elliston, B., Tromey, T. and Taylor, I. L. (2000). *GNU Autoconf, Automake and Libtool.* New Riders Publishing.

Wheeler, D. A. (2001). More Than a Gigabuck: Estimating GNU/Linux's Size. `http://www.dwheeler.com/sloc/`.

Winskel, G. (1993). *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts.

Wolf III, J. H. (1999). Programming Methods for the Pentium III Processor's Streaming SIMD Extensions Using the VTune Performance Enhancement Environment. *Intel Technology Journal.*