DIPLOMARBEIT

Proper Tail Recursion in C

ausgeführt am Institut für Computersprachen
der Technischen Universität Wien

unter Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall
und
Ass.Prof. Dipl.-Ing. Dr.techn. Ulrich Neumerkel

durch

Mark Probst

Trappelgasse 6/8
1040 Wien

Wien, am 2. Februar 2001

Abstract

The C programming language has been used successfully as the target language of compilers for numerous programming languages. However, compiling functional programming languages like Scheme and logic languages like Prolog to C is complicated by the lack of support for proper tail calls in C. Proper tail calls are function calls which reuse the caller's stack frame for the callee's. Many higher-level programming languages require that all function calls be implemented as proper tail calls.

This work explains why the standard C calling convention cannot support proper tail calls in the general case and presents a calling convention which supports proper tail calls for C, while retaining all features of ANSI-C function calls, especially variable argument functions, without incurring substantial overhead. On RISC machines, performance is even increased in some situations.

An implementation of this calling convention for the Alpha and the i386 architectures in the GNU Compiler Collection (GCC) is described and the generated code analyzed in detail. Furthermore, the problems with implementing this calling convention for the SPARC are described and their solutions sketched.

Zusammenfassung

Die Programmiersprache C wird erfolgreich als Zielsprache von Compilern für verschiedenste Programmiersprachen eingesetzt. Leider wird die Übersetzung funktionaler Sprachen wie Scheme und logischer Sprachen wie Prolog nach C erschwert, da C keine Proper Tail Calls zur Verfügung stellt. Proper Tail Calls sind Funktionsaufrufe, die den Stack Frame der aufrufenden Funktion für jenen der Aufgerufenen wiederverwenden. Viele höhere Programmiersprachen setzen voraus, daß alle Funktionsaufrufe als Proper Tail Calls implementiert werden.

Diese Arbeit erklärt, warum die übliche C Calling Convention Proper Tail Calls im allgemeinen Fall nicht unterstützen kann und präsentiert eine Calling Convention, die diese Eigenschaft hat, unter Beibehaltung aller Möglichkeiten von Funktionsaufrufen in ANSI-C, insbesondere Funktionen mit variabler Argumentzahl, ohne großen Aufwand zu verursachen. Auf RISC Maschinen ergibt sich in manchen Situationen sogar ein Geschwindigkeitszuwachs.

Es wird eine Implementierung dieser Calling Convention für die Alpha und die i386 Architekturen in der GNU Compiler Collection (GCC) beschrieben und der generierte Code im Detail anlysiert. Weiters werden die Probleme bei der Implementierung dieser Calling Convention für die SPARC beschrieben und deren Lösungen skizziert.

Contents

Chapter 1

Introduction

The C programming language has been used successfully as the target language of compilers for numerous programming languages. However, compiling functional programming languages like Scheme and logic languages like Prolog to C is complicated by the lack of support for proper tail calls in C. Proper tail calls are function calls which reuse the caller's stack frame for the callee's. Many higher-level programming languages require that all function calls be implemented as proper tail calls.

The lack of proper tail calls and other shortcomings of C have even motivated the design of a C-like portable assembler language called `C--`. [JNO98] lists proper tail calls as one of the features that C lacks as a portable assembler and advertises them as one of the main features of `C--`.

This work explains why the standard C calling convention cannot support proper tail calls in the general case and presents a calling convention which supports proper tail calls for C, while retaining all features of ANSI-C function calls, especially variable argument functions, without incurring substantial overhead. On RISC machines, performance is even increased in some situations.

An implementation of this calling convention for the Alpha and the i386 architectures in the GNU Compiler Collection (GCC) is described and the generated code analyzed in detail. Furthermore, the problems with implementing this calling convention for the SPARC are described and their solutions sketched.

1.1 Overview

Chapter 2 defines the notions of tail call, proper tail call and proper tail recursion. It motivates the need for the support for proper tail calls in the C programming language by describing some approaches to compiling functional and logic programming languages to C and pointing out their deficiencies.

Chapter 3 describes how the definition of a tail call from chapter 2 can be applied to the C programming language.

Chapter 4 describes how arguments are passed from caller to callee in the C programming language and the reasons behind this procedure. This is compared with how arguments are usually passed in Pascal implementations. It then describes in detail the calling conventions on Alpha, i386 and SPARC platforms and summarizes these together with calling conventions on the MIPS and PowerPC platforms.

Chapter 5 explains why the standard C calling convention does not support all cases of proper tail recursion and describes how the convention can be changed to support all circumstances.

Chapter 6 describes the actual implementation of this calling convention as a modification of the GNU Compiler Collection (GCC). Section 6.4 describes the tail call optimizations already done by GCC and points out their limitations. Architecture independent and architecture dependent problems are described and their solutions detailed. Section 6.7 summarizes all cases of function calls and epilogues that can occur and describes how the generated code handles them. Section 6.9 presents and analyzes generated assembler code for the Alpha and i386 architectures using the new calling convention.

Chapter 7 compares the performance between function calls using the standard C calling convention and the convention described in this work and analyzes the results.

Finally, chapter 8 concludes and presents directions of further work.

Chapter 2

Proper Tail Recursion

This chapter defines the notions of tail call, proper tail call and tail recursion. It motivates the need for the support for proper tail calls in the C programming language by describing some approaches to compiling functional and logic programming languages to C and pointing out their deficiencies.

2.1 Definition and Motivation

Definition tail call, proper tail call: A call from function f to function g is a *tail call* iff that function call is the last thing the function f does and nothing in f's activation record is needed during the execution of the call to g or after f's return. A tail call from f to g is a *proper tail call* iff f's activation record is freed before calling function g. Note that f and g need not necessarily be distinct.

Definition proper tail recursion: Proper tail recursion is the property of a programming language implementation of implementing all tail calls as proper tail calls.

The definition can easily be extended to a larger class of languages by substituting "procedure", "method", or "predicate" for "function".

Proper tail recursion is very important for functional and logic programming languages, since languages in these families often lack looping constructs and express loops through recursion.

As an example, let us implement the factorial function. The factorial $n!$ is defined as

$$n! = 1 \cdot 2 \cdot \ldots \cdot n = \prod_{k=1}^{n} k$$

It is quite natural to implement it in C as

```
int fac (int n) {
    int k, prod = 1;
    for (k = 2; k <= n; ++k)
        prod *= k;
    return prod;
}
```

In Scheme, it could be implemented as

```
(define (fac n)
  (let loop ((k 2) (prod 1))
    (if (> k n)
        prod
        (loop (+ k 1) (* prod k)))))
```

Although the call to `loop` looks like a function call (and in fact enjoys the same semantics), there exists at most one activation record of `loop` at any time. C functions corresponding to the Scheme code above would look like this[1]:

```
int fac (int n) {
    return loop(n, 2, 1);
}
int loop (int n, int k, int prod) {
    if (k > n)
        return prod;
    else
        return loop(n, k + 1, prod * k);
}
```

A C compiler is not required to make the tail calls to `loop` proper, i.e., calling `fac` with a big value for `n`, could lead to a stack overflow. This cannot happen in Scheme.

Most modern C compilers (including GCC [Sta99]) would implement the call from inside `loop` to `loop` itself as a proper tail call, but not the call from `fac` to `loop`.

Tail calls to functions other than the calling function itself are very common in languages like Scheme, ML or Prolog. Take, for example, this function, which is part of a Scheme interpreter written in Scheme [ASS96]:

[1]Since C, unlike Scheme, does not support closures, the environment (containing `n`) has to be passed explicitly.

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

The two last calls to `eval` are tail calls and must be implemented as proper tail calls.

Note that this function contains no tail calls:

```
int fac (int n) {
    if (n == 1)
        return 1;
    else
        return fac(n - 1) * n;
}
```

The recursive call to `fac` is not the last thing `fac` does, because after this call, it has to multiply its return value by `n`.

2.2 Implementing Functional and Logic Languages in C

There exist a lot of Scheme [KCR98] implementations which compile to C code, among them Bigloo [SW95], Gambit-C [Fee], Hobbit [Tam94], Scheme->C [Bar89], Stalin [Sis] and Chicken [Win].

Of these, the only properly tail recursive one is Chicken. It uses a technique suggested by [Bak95], which basically uses the C stack as a garbage collected heap. Function calls are always tail calls, but it is irrelevant whether those calls are implemented as proper tail calls by the C compiler. All Scheme data structures are allocated as local variables, i.e., on the stack. Before each function call, the stack is checked for overflow. If overflow is about to occur, a copying garbage collection is started, the roots being the arguments to the function about to be invoked and global variables. After finishing the garbage collection, the function is then called using the new heap as its stack, beginning at the end of the live data. Note that this technique does not depend on implementation details regarding stack layout and that it allows (and in fact requires) precise garbage collection, quite contrary to traditional compilation techniques to C. It does, however depend on a stack check for every function call, does a lot of unnecessary register saves and hence leaves a lot of obvious garbage on the heap (stack). However, see chapter 8 for suggestions on how to remedy this situation.

All other Scheme compilers mentioned suffer from their use of C function calls to implement Scheme calls, some of them being more clever than others about analyzing some cases of tail recursion and resolving them as jumps.

At least two ML compilers generating C code exist, namely sml2c [TLA92] and Camlot [Cri92], both of them being properly tail recursive. Proper tail recursion is achieved by using trampolines[2]. The function wishing to do a proper tail call returns the address of the function to be called (plus a pointer to an environment) to a driver function which just calls the next such function in a loop. Disregarding the environment, such a loop would look like this:

```
typedef void* cont (void);

for (;;) {
    cp = (cont*)(*cp)();
}
```

Usually, this loop is unrolled multiple times so as to minimize the overhead. Nevertheless, the overhead incurred by trampolines is still considerable, as described by Tarditi et al. [TLA92]. They close by suggesting two C compiler extensions to make C more efficient as a target for compilers for functional languages: global register variables and proper tail recursion. The former is already supported by GCC.

The idea of the trampoline appears earlier in [Ste78], which describes RABBIT, a compiler for Scheme which generates MacLisp code. Since the latter is not properly tail recursive, Steele uses what he calls the `UUO-HANDLER`, which appears to be the same in principle as the abovementioned trampolines.

Logic programming languages, especially Prolog, have also often been compiled to C, despite the difficulties involved. The first such system seems to be described in [WR88]. It generates relatively straightforward C code and does not treat tail calls specially, i.e., is not properly tail recursive, meaning that loops might fail due to stack overflow.

Other systems, like Janus [GBD92], compile a whole program into a single function. This function consists of one big `switch` statement, which acts as a dispatcher for indirect jumps. Each jump target has a `case` label, as well as a `goto` label. The latter is used for direct jumps. Indirect jumps cost two jumps plus one table lookup, since they must be dispatched through the `switch` (it is assumed that the C compiler implements the `switch` as a lookup table dispatch). However, C compilers tend to consume tremendous

[2] [Cri92] does not mention that this technique is used. I verified that it is by trying out Camlot.

amounts of memory and to be very slow at compiling large functions, so this approach becomes impractical for larger programs. Furthermore, it prevents separate compilation.

Due to these reasons, [DM94] argues for a combination of the latter technique with the use of trampolines. Predicates calling each other frequently could be compiled into single C functions. Calls to other predicates would involve returning from that function the data needed to identify the predicate to call: the function of that predicate and the right entry point within it.

KLIC [FCRN94], which is an implementation of KL1, takes that approach. Every module is compiled into a single C function. Extra-module calls, as well as indirect calls which cannot be proven to be intra-module, are done through a trampoline. This approach seems reasonable. However, let us examine the costs of the various types of jumps:

Type	Cost
Intra-module, direct	1 direct jump
Intra-module, indirect	1 direct jump, 1 table lookup, 1 indirect jump
Extra-module	1 function return, 1 function call, 1 table lookup, 1 indirect jump

Another approach is taken by Turbo Erlang [Hau94] and Mercury [HCS95]. They use GCC's first class labels feature to make inter-procedural `goto`s. Though effectively reducing each jump to a jump on the target machine, it has the disadvantage of being unportable (since it depends on certain properties of the generated code, which could change in any release of the compiler), not allowing local variables in the C functions and not working on some architectures, like the Alpha (see [HCS95] for details).

A very similar technique is used by `wamcc` [CD95]. Instead of using GCC's first class labels, it uses inline-assembler labels at the beginning of each function and instead of `goto`s uses function calls to jump to these labels. It suffers from the same disadvantages as the first class labels approach.

2.3 Motivation

The previous section has shown that the provision of proper tail calls in C would be a big help for implementations of functional and logic programming languages, making obsolete a large number of often unportable and inefficient kluges.

Another area where proper tail calls might prove beneficial are threaded code interpreters. Direct threading is impossible to implement in ANSI

C [Ert95], but can be implemented in GNU C through the use of first-class labels. The disadvantages are, analogous to the Janus approach, longer compilation times and the impossibility of separate compilation. Proper tail calls would solve these problems.

Finally, one could speculate that the availability of proper tail calls in C might have a positive effect on the coding style of C programmers. Baker argues [Bak97] that proper tail recursion makes it easier to write clear and elegant programs which are also efficient.

All the mentioned motivations have in common that they require that whether or not a call is implemented as a proper tail call is predictable. A compiler for a programming language generating C code must be able to rely that the C compiler generates proper tail calls, especially if the source language requires proper tail recursion. The implementor of a direct threaded code interpreter must know the circumstances under which the call to the next instruction is implemented as a proper tail call. In other words: The C compiler must guarantee that under certain well defined circumstances calls are always implemented as proper tail calls. Furthermore, these circumstances should be easy to understand and should be general enough to be easy to fulfill.

Chapter 3

Proper Tail Recursion in C

This chapter applies the definition of a tail call from the previous chapter to the C programming language, closing with a definition of a tail call in C.

3.1 Tail Calls in C

According to our definition of a tail call in section 2.1, a function call has to fulfill two requirements in order to be a tail call:

- It has to be the last thing the calling function does.
- The calling function's activation record must not contain anything that is needed during the execution of the call or afterwards.

3.1.1 Syntax

The first requirement can almost be resolved syntactically. We will take an approach similar to [KCR98]: We define the places in which a function call can be a tail call. Additionally, we require that the return types of the caller and the callee match. If they do not, an implicit type cast must be performed after the callee returns, hence the call is not the last thing the caller has to do.

In order to do this we extend the C grammar given in the C99 standard [Ame99] by productions that are similar to those already in the grammar, but are annotated with "tail" suffices. A new nonterminal, *function-call*$_{tail}$, denotes a function call which may be a tail call. Note that such a function call must also fulfill the second requirement in order to be a tail call.

All new productions and all changes to the C grammar are given in appendix A. This section only presents the interesting parts.

A function definition has the following syntax (the “opt” suffix describes optional parts):

function-definition:
  declaration-specifiers declarator declaration-list$_{opt}$ *compound-statement*$_{tail}$

If nothing remains to be done after the execution of an `if` statement as a whole, then that holds for its two branches as well. The `switch` statement is much more complicated, due to C’s liberal syntax for this statement. If a `switch` statement is tail-annotated, then we must tail-annotate within its body the last statement and every statement which is directly followed by a `break` statement.

selection-statement$_{tail}$*:*
  `if (` *expression* `)` *statement*$_{tail}$
  `if (` *expression* `)` *statement*$_{tail}$ `else` *statement*$_{tail}$
  `switch (` *expression* `)` *statement*$_{tail}$

The `switch` statement requirement is implemented in *compound-statement*$_{tail}$ and friends, whose definitions are given in appendix A. The definitions are as complicated as they are because we avoid ambiguities in the grammar. These additions do not interfere with `break` statements in loops, since iteration statements are never tail-annotated.

Since we must also tail-annotate the expressions of `return` statements under all circumstances, we must modify the *jump-statement* definition of the original grammar:

jump-statement:
  `goto` *identifier* `;`
  `continue ;`
  `break ;`
  `return` *expression*$_{tail\ opt}$ `;`

Since the comma operator is specified as first executing its left and then its right operand, the latter can be tail-annotated if the expression itself is:

expression$_{tail}$*:*
  assignment-expression$_{tail}$
  expression `,` *assignment-expression*$_{tail}$

The argumentation for making both branches of the conditional expression tail-annotated is the same as for the `if` statement:

conditional-expression$_{tail}$:
  logical-OR-expression$_{tail}$
  logical-OR-expression `?` *expression*$_{tail}$ `:` *conditional-expression*$_{tail}$

Although C uses short-cut evaluation for logical AND and OR, we cannot make the right operand of these operands tail-annotated. The reason is that the standard specifies that the result of the operation be `1` if it is logically true, i.e., `a&&b` is equivalent to `a?(b?1:0):0` and `a||b` is equivalent to `a?1:(b?1:0)`. This “conversion” must be performed after a function call.

logical-OR-expression$_{tail}$:
  logical-AND-expression$_{tail}$
  logical-OR-expression `||` *logical-AND-expression*

logical-AND-expression$_{tail}$:
  inclusive-OR-expression$_{tail}$
  logical-AND-expression `&&` *inclusive-OR-expression*

Now we finally get to where *function-call*$_{tail}$ can be expanded:

postfix-expression$_{tail}$:
  primary-expression$_{tail}$
  postfix-expression `[` *expression* `]`
  function-call$_{tail}$
  postfix-expression `_` *identifier*
  postfix-expression `->` *identifier*
  postfix-expression `++`
  postfix-expression `--`
  `(` *type-name* `)` { *initializer-list* }
  `(` *type-name* `)` { *initializer-list* `,` }

function-call$_{tail}$ looks exactly like a normal function call. After all, we just wanted to see where we can find it, not define a new syntax for it:

function-call$_{tail}$:
  postfix-expression `(` *argument-expression-list*$_{opt}$ `)`

It is possible to construct situations in which a function call would be a tail call according to our definition (see section 2.1) but would not satisfy our syntax requirement. A very simple example:

```
void f (void) { g();; }
```

The call to `g` is a tail call, because the empty statement does nothing. The same goes for statements like `1+1;`. As a consequence, our syntactic requirement cannot catch all cases, although it will catch most non-contrived tail calls. On the other hand, one could argue that executing the empty statement is doing something, albeit its producing no side-effect. Function calls which could be implemented as tail calls but do not satisfy the syntactic requirement can also be written in Scheme, for example:

```
(let ((val (f)))
  val)
```

This is of course semantically equivalent to `(f)`.

3.1.2 Activation Record

The second requirement states that the calling function's activation record must not contain anything that is needed during the execution of the call or afterwards.

Since a tail call is a function's last action and its activation record is destroyed after its return, what happens if something in the activation record is needed after the tail call is undefined anyway, so we need not concern ourselves with this case.

It remains to be investigated in which cases something in the activation record of the caller is needed during the execution of the callee. An activation record of a C function contains:

- Incoming arguments
- Saved registers
- The return address, possibly in the form of a saved register
- Local variables and spilled registers

Saved registers must be restored before a tail call, whereas the return address becomes part of the callee's activation record, if it is not a saved register. What remains are incoming arguments and local variables (spilled registers can be treated like local variables in this context). Since C only supports call-by-value, these can only be accessed if their address is saved somewhere not in the activation record of the caller. This can be

- The arguments to some function called from within the caller

- Global variables
- `static` variables within the caller
- Some other memory location known not only by the caller (like some memory location on the heap)

3.2 Conclusion

Definition tail call in C: A function call in C is a tail call iff the following three criteria are fulfilled:

- The call site is a $\textit{function-call}_{tail}$ according to our extended C grammar.
- The return types of the caller and the callee match.
- The function containing the call site does not store the address of an incoming argument or a local variable, or something from which such an address could be constructed, into some lvalue which is visible outside the caller's invocation. The only lvalues which are not potentially visible outside that invocation are incoming arguments and local variables of the caller. Note that passing a value as an argument to a function counts as an assignment.

Chapter 4

Calling Conventions for C

The function calling pattern of the C language is typical for procedural languages: Function invocations can be nested arbitrarily, including direct and indirect recursion, and invocations finish in the reverse order of their starting. The `longjmp` function can be seen as a straightforward extension to this pattern, in that it allows finishing more than one invocation in one step, but without violating the last-in-first-out principle, hence it need not concern us here.

The described calling pattern suggests that the activation records be allocated on a stack, which is what virtually all C compilers do. It is theoretically possible to use other allocation strategies, like allocating activation records on the heap, but this is not done, because of the added complexity and performance losses.

Languages like Scheme and ML depart from the last-in-first-out principle, in that function invocations can be saved and restored later on, allowing functions to exit more than once, like predicates in backtracking languages like Prolog.

The most important question for us is where the arguments for a function are placed. The fastest way is obviously to place them into registers, which is exactly what C compilers do on RISC architectures like the Alpha. However, it is necessary to account for an unlimited number of arguments and register sets are always very small. Hence, there needs to be some place to store arguments which do not fit into the register set.

One possibility would be to use a large enough fixed region of memory where outgoing arguments are stored. This has the disadvantage that arguments for a function `f` would have to be copied from that region into the activation record if `f` called `g` and after that call referred to its arguments:

```
int f (int a) {
  int b = g();
  return a + b;
}
```

If `a` were stored in that fixed region, the call to `g` might disrupt its value. Hence, `f` would have to copy it into its activation record first, which means that it would have to be copied twice: First, by the caller of `f` to the argument region and then by `f` into its activation record. The reasons why this copying is not an overhead for register arguments—which need to be copied to the activation record to be preserved, as well—are that copying a value into a register is cheaper than copying it into memory and that—at least on RISC machines—it must be copied into a register before it can be written to memory, anyway.

The obvious solution is to place the arguments on the stack before calling the function. This is indeed what virtually all C compilers do.

The next question is the order in which the arguments are to be placed on the stack. The two obvious alternatives are

- Push the arguments in the order they are specified, i.e., the last argument is on the top.
- Push the arguments in reverse order of specification, i.e., the first argument is on the top.

Pascal compilers [PD82] have traditionally pushed arguments in the order of their specification. C compilers, on the other hand, use the reverse order.

The reason for this lies in the presence of variable argument functions. Functions in C can be declared to take some fixed arguments plus any number of additional arguments, the types of which can be arbitrary. Of course, the caller and the callee must agree on the number and types of the arguments, which is usually done through one of the fixed arguments, like the format string in a `printf` function call. The C compiler however, does not and cannot know the number and types of the variable arguments of a function. Were C to use the Pascal way of pushing arguments on the stack, the fixed arguments (which always come before the variable arguments) would have an unknown offset from the top of the stack. This could of course be remedied by pushing as an implicit argument for example the length of all variable arguments on the stack. It is much easier, however, to just reverse the order of arguments on the stack, thereby making the fixed arguments begin at a known offset from the top of the stack.

	Pascal	C
Order in which arguments are pushed	Syntactic order	Reverse syntactic order
Arguments are popped by	Callee	Caller
Variable argument functions possible?	No	Yes

Table 4.1: Comparison between Pascal and C calling conventions

Another import question is where the arguments are popped off the stack. This can be done either in the caller or in the callee. In Pascal, where arguments are fixed, both solutions are viable, because both the caller and the callee know how much argument space has been popped. Pascal implementations tend to do it in the callee. The i386 architecture even has a `ret` instruction version which performs the function return and the popping in one step. In C, variable argument functions do not know the amount of arguments that has been pushed, so they cannot pop it. Hence, C implementations pop the arguments in the caller.

Table 4.1 shows a comparison between the Pascal and C calling conventions as described here. Note that this is not a comparison between the two languages but between the calling conventions their implementations usually use.

4.1 Specific Calling Conventions

The following subsections describe in detail calling conventions for the Alpha, the i386, and the SPARC. These are followed by a summary of these three and MIPS and PowerPC calling conventions. The calling convention descriptions are in prose, which is common practice. It should be noted, though, that Bailey and Davidson have developed a formal language [BD95] for the specification of calling conventions.

The term "structure" in the following subsections refers to both C structs and unions.

4.1.1 Alpha

This section describes the standard calling convention used on the Alpha [Com00b, Com00a] under Tru64 UNIX (the operating system formerly known as Digital UNIX (the operating system formerly known as Digital OSF/1)) and Linux.

Name	Use	Callee-saved?
`$0`	Holds function results. Otherwise a temporary register.	No
`$1`–`$8`	Temporary registers.	No
`$9`–`$14`	Saved registers.	Yes
`$15` (`$fp`)	Used as the frame pointer if it is needed. Otherwise a saved register.	Yes
`$16`–`$21`	Argument registers.	No
`$22`–`$25`	Temporary registers.	No
`$26` (`$ra`)	Return address for calls.	Yes
`$27` (`$pv`)	Procedure value.	No
`$28`	Reserved for the assembler.	No
`$29` (`$gp`)	Global pointer.	No
`$30` (`$sp`)	Stack pointer.	Yes
`$31`	Always zero.	n/a
`$f0`	Holds function results. Otherwise a temporary register.	No
`$f1`–`$f9`	Saved registers.	Yes
`$f10`–`$f15`	Temporary registers.	No
`$f16`–`$f21`	Argument registers.	No
`$f22`–`$f30`	Temporary registers.	No
`$f31`	Always zero.	n/a

Table 4.2: Conventional register usage on the Alpha

The Alpha has 32 integer registers (`$0`–`$31`) and 32 floating-point registers (`$f0`–`$f31`). Table 4.2 summarizes conventional register usage.

All code on the Alpha is position independent by convention. To address static constants (like large integer constants, floating point constants, function addresses, string constants, ...) a so called GOT (global offset table) is used. It contains the addresses of these constants. Since the Alpha allows register-relative addressing only with 16 bit offsets, the GOT is divided into segments, each of which can be up to 64K long. Functions in the same compilation unit share the same segment, but functions from different compilation units may use different segments. The address of the GOT segment of the current function is stored in the global pointer (register `$gp`). This address is determined relative to the current function. To this end, the code for a function call stores the address of the function to be called in the procedure value register (`$pv`) and performs a jump to that location. The callee then sets the global pointer relative to the procedure value. A similar thing happens after a function returns to its caller. If the caller cannot be sure that the callee uses the same GOT segment, it must reload the global pointer. It does this relative to the return address register (`$ra`). This means that a function return must always be implemented as a jump to the address in the return address register.

Arguments are passed in registers and on the stack. The first six arguments are passed in registers: Argument n (with n from 1 to 6) is passed in integer register $16 + (n - 1)$ if it is an integer, pointer or structure argument. If it is a floating point argument, it is passed in floating point register $16+(n-1)$. Arguments from 7 onwards are passed on the stack beginning at the address the stack pointer points to, in the direction of increasing memory addresses. Each argument is aligned on an 8 byte boundary: Argument n (with $n > 6$) is passed in memory location $\texttt{\$sp}+8(n-7)$. Arguments smaller than 8 bytes are extended to that size. Unsigned 8 and 16 bit integers are zero-extended, signed integers and 32 bit unsigned integers are sign-extended and the contents of the 32 unused bits of floating point numbers are undefined.

Structures are handled as integer parameters, even if they contain floating point elements. A structure of length $8n$ is treated as n consecutive integer arguments. This means that structures may be passed completely in consecutive registers, on the stack or even partly in consecutive registers and partly on the stack.

Integer and pointer values are returned by a function to its caller through the register `$0`. Floating point results are returned through `$f0`. Structure return values are handled differently: The caller has to reserve space for the structure to be returned by the callee (usually in the caller's stack frame) and pass the address of this block as an implicit first argument. The callee stores the result structure there. This applies regardless of the size of the structure.

The stack pointer always has to be aligned on a 16 byte boundary.

4.1.2 i386

There are a lot of calling conventions in use for the i386 architecture. GCC, for example, allows some variations, like passing up to three arguments in registers. I describe here the one used in Linux by default. Table 4.3 summarizes integer register usage in this convention.

All arguments are passed on the stack, beginning with the first argument at location $\texttt{\%esp}+4$, occupying increasing memory addresses. All arguments must be aligned on 4 byte boundaries. The return address must be stored in location `%esp`. This is usually handled by using the `call` instruction,

Name	Use	Callee-saved?
`%eax`	Holds function results. Otherwise a temporary register.	No
`%ebx`, `%esi`, `%edi`	Saved registers.	Yes
`%ecx`	Temporary register.	No
`%edx`	Holds the high 32 bits of 64 bit function results. Otherwise used as a temporary register.	No
`%esp`	Stack pointer.	Yes
`%ebp`	Frame pointer. May be unused.	Yes

Table 4.3: Conventional register usage on the i386

which decrements `%esp` by 4, stores the address of the next instruction in that location and performs a jump to an address given as its argument.

Integer and pointer function results are returned through the `%eax` register[1]. Floating point results are returned through the top floating point register ST(0) [Int00]. If the callee is to return a structure, the caller has to reserve space for it (usually in its stack frame) and pass the address of this block in an implicit first argument to the callee, which stores the structure it returns there.

The floating point stack must be considered clobbered by a function call (with the exception that the top floating point register contains a floating point result if the callee returns one).

4.1.3 SPARC

This section describes the standard calling convention used on the SPARC V8 [SPA92, The].

The SPARC differs from other common RISC architectures in its provision of so-called register windows, which makes its calling convention unique among the common architectures.

At any time, there are 32 integer registers accessible to the application. 8 of them are global (`%g0`–`%g7`), meaning that they are always mapped to the same CPU registers, just like ordinary registers on other architectures. The other 24 integer registers are mapped to the current register window. A SPARC implementation can have from 2 to 32 register windows.

[1] `long long` function results (64 bit integers) are returned with the low part in `%eax` and the high part in `%edx`

Let us assume an implementation with n register windows, numbered from 0 to $n-1$. A register window consists of 8 local registers, 8 in registers and 8 out registers. Let us call these registers $\text{local}_{i,j}$, $\text{in}_{i,j}$ and $\text{out}_{i,j}$, with i from 0 to $n-1$ denoting the register window and j from 0 to 7. The in registers overlap with the out registers of the next higher register window, where we define the next higher register window of window $n-1$ to be window 0, i.e., registers $\text{in}_{i,j}$ and $\text{out}_{i+1 \bmod n,j}$ are the same physical register. All other registers do not overlap.

Formally: Registers $f_{i,j}$ and $g_{k,l}$ overlap (are the same register) iff $(f = g \wedge i = k \wedge j = l) \vee (f = \text{in} \wedge g = \text{out} \wedge k = i + 1 \mod n \wedge j = l) \vee (f = \text{out} \wedge g = \text{in} \wedge i = k + 1 \mod n \wedge j = l)$.

To determine which register window is currently mapped to the register set, the SPARC has a so-called current window pointer (CWP), which can take on values from 0 to $n-1$. The registers `%i`j, `%l`j and `%o`j are mapped to the physical registers $\text{in}_{\text{CWP},j}$, $\text{local}_{\text{CWP},j}$ and $\text{out}_{\text{CWP},j}$, respectively, with j from 0 to 7.

This setup is used by the calling convention to minimize the need for saving registers to the stack in user code. Table 4.4 summarizes conventional register usage on the SPARC.

The callee-saving of the in and local registers is automatically taken care of by the register windows. In the prologue, a function issues a `save` instruction, which decrements the current window pointer (or sets it to the highest value if it was 0). This makes the out registers become the in registers and the previous in registers disappear. Furthermore, the old local registers disappear and a new set becomes available. This automatically makes the old stack pointer become the new frame pointer. Since `save` can also perform an addition, the new stack pointer is also set with this instruction. The reverse operation is performed in the epilogue by the `restore` instruction.

Since a SPARC implementation only has a very limited number of register windows and a much higher number of function activations must be supported, it is clear that the contents of register windows must sometimes be written to memory. This is why the stack pointer (in register `%sp`) must always point to a region of 64 bytes which is not used by user code. It may be used by the system to store the local and in registers of a register window when and if that is necessary. If such an action is taken by the system, the local and in registers of the register window to be saved are saved in the block pointed to by the stack pointer of that register window. More formally: Saving register window i means storing registers $\text{in}_{i,j}$ and $\text{local}_{i,j}$ with j from 0 to 7 in the 64 bytes pointed to by $\text{out}_{i,6}$.

The stack pointer must always be aligned on an 8 byte boundary.

Arguments are passed in registers `%o0`–`%o5` and on the stack. Floating

Name	Use	Callee-saved?
`%g0`	Always zero.	n/a
`%g1`	Temporary register.	No
`%g2`–`%g4`	Global register.	n/a
`%g5`–`%g7`	Reserved for system.	n/a
`%i0`	Incoming argument register, outgoing function result.	Yes
`%i1`	Incoming argument register. Holds the low 32 bits of outgoing 64 bit function results.	Yes
`%i2`–`%i5`	Incoming argument registers.	Yes
`%i6` (`%fp`)	Frame pointer.	Yes
`%i7`	Return address − 8.	Yes
`%l0`–`%l7`	Saved registers.	Yes
`%o0`	Outgoing argument register, incoming function result.	No
`%o1`	Outgoing argument register. Holds the low 32 bits of incoming 64 bit function results.	No
`%o2`–`%o5`	Outgoing argument registers.	No
`%o6` (`%sp`)	Stack pointer.	Yes
`%o7`	Outgoing return address − 8. Otherwise a temporary register.	No
`%f0`	Holds function results. Otherwise a temporary register.	No
`%f1`–`%f31`	Temporary registers.	No

Table 4.4: Conventional register usage on the SPARC

point arguments are treated as integer arguments: A single precision floating point argument as one integer argument and a double precision floating point argument as two consecutive integers, the first containing the high 32 bits, the second the low 32 bits of the floating point value. Structures are passed by reference, meaning that the caller must make a copy of the structure to be passed somewhere (usually in its own stack frame) and pass its address as an integer argument. The copy must be made since the callee is allowed to change the structure pointed to by the argument, but C requires that structures be passed by value.

Integer and pointer return values are passed by the callee to the caller through the `%o0` register[2]. 64 bit integer registers are returned through register `%o0` and `%o1`, the former holding the high 32 bits, the latter the low 32 bits. Floating point return values are passed through register `%f0`. Structures are returned by copying them to some location determined by the caller (usually in the caller's stack frame), the address of which is passed on the stack.

The first six arguments are passed in registers `%o0`–`%o5`, using `%o0` to store the first argument and `%o5` to store the sixth. As already mentioned, the stack pointer must always point to a region of 64 bytes not used by user code, entry to a function being no exception. The next 4 bytes (at `%sp` + 64) are reserved for the address a structure return value should be copied to. If the callee does not return a structure, these 4 bytes are unused, but must be reserved nevertheless. The next 24 bytes (at `%sp` + 68) may be used by the callee to store its incoming register arguments, in order to make variable argument list processing easier. They must be reserved by the caller in any case. Following that (at `%sp` + 92) are arguments past the sixth, increasing syntactic order corresponding to increasing memory addresses. All stack arguments must be aligned on 4 byte boundaries.

Returning structures in the SPARC standard calling convention is a topic in its own right. The pointer to the memory block a structure to be returned is to be copied to at `%sp` + 64 is only one part of the story. The other is that if the caller wants the callee to copy the structure to that location, the next but one instruction after the `call` to the function, i.e., the instruction at the location `%o7` + 8 upon entry to the function, must be an `unimp` instruction, the immediate operand of which must be the low-order 12 bits of the size of the structure to be returned. Otherwise, that instruction can be any ordinary instruction. Before the callee copies the structure into its destined location, it checks whether the instruction at the return address is an `unimp`

[2]Note that the callee has to store the value in `%i0` if it uses the `restore` instruction afterwards to switch register windows.

instruction. If it is and if its operand matches the low-order 12 bits of the structure the callee is about to return, the callee copies the structure and returns to the instruction after the `unimp`. Otherwise, it returns normally, but without copying, meaning that if the `unimp` instruction is not present, the structure is not copied, which of course only makes sense if the caller is not interested in the return value. It also means that if the caller and the callee disagree about the low-order 12 bits of the size of the structure, control is transferred to the `unimp` instruction, thereby generating a trap, usually resulting in a core dump.

It is also worth mentioning that while GCC on the SPARC generates the `unimp` instruction in the caller, it does not check for this instruction in the callee, i.e., it always copies the structure and always returns to the next but one instruction after the call. In other words, it assumes that the `unimp` instruction is present in the caller and that the low-order 12 bits of the sizes match.

4.1.4 Summary

Table 4.5 contains a summary of the three calling conventions described in the previous subsections. In addition, it summarizes the following two calling conventions:

- The N64 calling convention for the MIPS, as described in [Sil96].
- The System V calling convention for the PowerPC, as described in [SI95].

The following items are the same on all summarized calling conventions:

- Integer and floating point return values are always passed in registers. Integer return values always in integer registers, floating point return values always in floating point registers.
- The stack grows downward, i.e., towards lower memory addresses.

		Alpha	MIPS N64	PowerPC System V	SPARC V8	i386
Architecture specifics	Word size in bits	64	64	32	32	32
	General purpose integer registers	31	31	32	39–519 (31 visible at any time)	6
	General purpose FP registers	31	32	32	32	8 (arranged stack-like)
	Hardware support for function calls	none	none	return address register	register windows	stack- and frame-pointer registers, `call`, `ret` instructions
Integers	Saved integer registers	9	11	19	17	5
	Integer argument registers	6	8	8	6	0
Floating points	Saved FP registers	9	8	18	0	0
	FP argument registers	6	8	8	0	0
	FP arguments passed in integer registers	never	only in variable part of vararg function	never	always	n/a
	Independent integer/FP argument register numbering	no	no	yes	n/a	n/a
	`floats` promoted to `doubles`	no	no	yes	no	no

		Alpha	MIPS N64	PowerPC System V	SPARC V8	i386
Structures	Structures passed by	value	value	reference	reference	value
	Structure elements passed in FP registers	never	double FPs if not unioned	never	never	never
	Structures returned in register(s)	never	up to 128 bits (2 registers)	up to 64 bits (2 registers)	never	never
	Structure value address passed as/in	implicit first argument	implicit first argument	implicit first argument	designated location on stack	implicit first argument
Stack	Stack alignment in bytes	16	16	16	8	4
	Argument alignment in bytes	8	8	4 (8 for FP arguments)	4	4
	Stack usage in bytes for recursive function without arguments (per invocation)	16	16	16	96	4
	Circumstances under which minimum frame size is fully utilized (least space wasted)	one word of local variable space or register save area on the stack	one word of local variable space or register save area on the stack	two words of outgoing parameter space, local variable space, `CR` register save area or register save area on the stack	variable argument function taking using six argument registers and returning a structure, all local and in registers live at outgoing call site	always
	Return address passed in/on	register	register	special register (`LR`)	stack	stack

Table 4.5: Calling conventions summary

Chapter 5

A C Calling Convention for Proper Tail Calls

In order to determine why the C calling convention is not suitable for proper tail calls, let us assume a calling conventions where all arguments, including the return address, are passed via the stack, like on the i386. A function call `f(1,2)` would look something like this:

```
x:
  push 2
  push 1
  push x_ra
  jmp f
x_ra:
  pop
  pop
  pop
```

We see here the two main characteristics of the C calling conventions: Arguments are pushed in reverse order and are popped by the caller. Now let's say the function `f` is defined as follows:

```
void f (int i, int j) {
    g(3);
}
```

Using the C calling convention, this would be compiled into something similar to this:

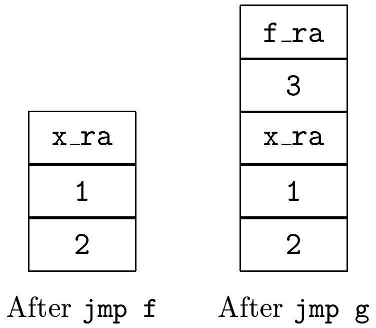

Figure 5.1: Stack contents for the execution of the call to `f`

```
f:
  push 3
  push f_ra
  jmp g
f_ra:
  pop
  pop
  jmp 0(sp)
```

We assume that `sp` is the stack pointer and `0(sp)` the word pointed to by it (offset `0`).

Figure 5.1 shows the contents of the stack during the execution of the call to `f`. The call to `g` is, though being a tail call, obviously not a proper tail call, since the arguments to `f` remain on the stack during the execution of `g`.

Let us now try to convert the call to `g` to a proper tail call. What do we have to do in order to accomplish this? First, we need to get rid of `f`'s activation record. In this case, it contains the arguments `i`, `j`, and the implicitly passed return address. We can just delete `i` and `j`, but we need to remember the return address since it is the address `g` has to return to after it has completed execution. Then we have to push the arguments to `g`, including the return address and do the call:

```
f:
  pop r1
  pop
  pop
  push 3
  push r1
  jmp g
```

We assume that `r1` is a temporary register. The effect of this code is that `g` returns directly to `x`, from which `f` was called. Now let us look at what happens when `g` returns to `x`. After the call to `f` is completed, `x` pops the arguments it pushed for `f`, which are three. However, the function returning to `x`, namely `g`, has been passed only two arguments, which means that `x` pops too much. Similarly, had the function `g` taken more than three arguments, `x` would have popped too little. This results from the simple fact that when proper tail calls are possible, the function returning to the point after a call site is not necessarily the function that was called at that call site. In the general case, the compiler cannot know which function will return, and hence cannot know how many arguments it would have to pop. For example, assume `f` were defined thus:

```
void f (int i, int j) {
    if (rand() % 2) g(3);
    else h(4, 5, 6);
}
```

The function returning to `x` might now be either `g`, `h` or a function called from within one of these functions.

It is possible to solve this problem in two different ways:

- Let someone else instead of the caller pop the arguments.
- Make the function returning to the caller report how much argument space needs to be popped.

The first alternative more or less implies that the callee pop the arguments, like in the Pascal calling convention, while the second one departs only little from the C calling convention. It is clear, however that returning a second value from every function is more overhead than just letting the callee pop the arguments. There is another catch to the two solution, though. In the case of variable argument functions, the callee does not know what amount of arguments it has been passed, and hence cannot pop them or return their amount to the caller. It is thus clear that we have to do some extra work for variable argument functions.

We have already argued in chapter 4 that the compiler cannot know the amount of arguments a variable argument function has been passed when the C calling convention is used. This information is only available at the call sites. Since we demand that the callee pop the arguments, we must transport the information from the caller to the callee, i.e., we need an additional, implicit argument. Although it seems obvious that we must use this argument to pass the amount of arguments to the callee, there is another possibility: Pass the address that the stack pointer has to be restored to, i.e., the address of the bottom of the outgoing arguments on the stack, which I call "bottom pointer". In order to compare the two in terms of efficiency, two operations have to be taken into account:

- Calculating the actual value when performing a call to a variable argument function. This is a null-operation for the argument block size approach, since the amount of arguments is a constant known at compile time. For the bottom pointer approach, it can be calculated as the stack pointer plus a constant known at compile time. This is a win of one instruction for i386, but makes no difference on RISC machines like the Alpha, which can add a register and a constant and move the sum to another register in one instruction.

- Calculating the address at which an outgoing argument for a proper tail call has to be placed. This is just the sum of the bottom pointer and a constant known at compile time for the bottom pointer approach, but it is more complicated when the argument block size is used. For that case, the address is the sum of the stack pointer, the argument block size and a compile-time constant, which costs at least one instruction more than the bottom pointer calculation.

We conclude that the argument block size solution is at least as expensive as the bottom pointer approach both for i386 and RISC machines, while the latter is cheaper on RISC machines. Consequently, I have implemented the bottom pointer solution.

The bottom pointer is only passed to variable argument functions. Should it be necessary to allow calls to fixed argument functions with more actual than formal arguments (the behavior of such a call is undefined in ANSI C), the bottom pointer would have to be passed to and used in fixed argument functions as well.

I will refer to this calling convention with the name "proptail".

Chapter 6

Implementation

In the preceding chapter we have altered the C calling convention in order to make proper tail calls possible. This chapter describes the technical problems to be faced in implementing it and outlines an actual implementation in GCC.

6.1 The Approach

Since it was the goal of this work not only to produce a properly tail recursive implementation of C and thereby demonstrating its feasibility, but also to make it an attractive option for implementors of functional and logic languages to use, it was clear from the onset that it would be best to aim for this implementation to be included in the standard distribution of GCC.

Changing the calling convention GCC uses by default was clearly not an option. A change which breaks as much code as this would do would have to provide very significant advantages in major areas to be accepted into the GCC distribution, and to most of the GCC maintainers proper tail recursion is (quite understandably) a side issue at best.

The approach was therefore to establish a second calling convention. Functions to be called using this convention have to be syntactically marked accordingly. Of course, it should be possible to mix calling conventions.

6.2 GCC

The GNU Project's [Fou] GNU Compiler Collection (GCC) [Sta99], which was originally called GNU C Compiler, is a portable and optimizing compiler for several programming languages, most prominently C.

GCC uses two intermediate representations:

- Tree code is a target independent representation, roughly corresponding to C syntax trees. It is usually generated by the parser of the source language.

- Register Transfer Language (RTL) code is the representation on which most of the optimizations operate. It describes the effects of target machine instructions, which makes RTL code target dependent, even though RTL itself is not, since it operates on a lower level. RTL code is usually generated from tree code.

GCC compiles from a source language (the standard distribution includes front-ends for C, Objective-C, C++, Fortran, Chill and Java byte-code) to assembler for the target machine. Compilation is done in several passes:

- Parsing. As already mentioned, this translates the source language program to tree code, after type-checking. Constant folding and some arithmetic simplifications are done here.

- RTL generation. For the C language, this is actually not a separate pass from parsing. Inlining decisions happen in this pass, as well as tail recursion detection and optimization.

- Several optimization passes, such as jump optimization, common subexpression elimination and loop optimization.

- Data flow analysis. This identifies basic blocks and computes liveness of pseudo-registers.

- Instruction combination and register movement passes. The former tries to combine primitive instructions into more complex, hopefully faster ones. The latter tries to avoid register to register moves caused by instructions requiring values in specific registers by changing the register in question.

- Instruction scheduling.

- Register allocation, which consists of four passes: Register class preferencing, which determines the register class for each pseudo register; local register allocation, which determines hard registers for pseudo registers used only in single basic blocks; global register allocation which tries to assign hard registers to all remaining pseudo registers; reloading, which assigns stack slots to all pseudo registers, which did not get hard registers and produces the required loads and stores.

- Instruction scheduling. This pass is repeated here to take into account the spills for pseudo registers, which ended up on the stack.
- Jump optimization.
- Delayed branch scheduling, which tries to place instructions into delay slots of branches, and branch shortening, which determines which sorts of jumps can be used on which occasions.
- Conversion from traditional register usage to usage of a register stack, which is only needed for the Intel i386's FPU.
- Final. This outputs the assembler code, while performing peephole optimizations.
- Debugging information output.

6.3 Syntax

GCC already has a way of specifying non-standard attributes in declarations, namely with the `__attribute__` keyword. It is followed by an attribute name with optional arguments, within double parentheses. Since we only need a way to flag a function, we don't need any arguments:

```
extern __attribute__((tailcall)) int func (int);
```

Functions can also be defined with this attribute:

```
__attribute__((tailcall)) int func (int x) {
  ...
}
```

Of course, it is also necessary to flag pointers to functions with the attribute if the functions it points to use the new calling convention:

```
__attribute__((tailcall)) int (*fptr) (int) = 0;
```

Note that the compiler will assume undeclared functions to use the standard calling convention.

Some people have suggested that it would be advantageous to have a syntax for function calls which guarantees that the function call in question is implemented as a proper tail call. In cases where this is not possible, the compiler would give an error or at least issue a warning. I have not implemented such a syntax because I believe that the criteria for proper tail calls are easy enough to understand and because I doubt that such a syntax would be accepted into GCC, especially in the light that it does not add any semantic content to the language: The same code would be generated for proper tail calls regardless which syntax they use. The only difference would be that calls which are not implemented as proper tail calls would cause the compiler to complain with the new syntax. The sole use of such a syntax would be as a debugging aid.

6.4 The Existing Tail Call Optimization

GCC already has an optimization for certain cases of tail calls. First of all, the optimization must honor the standard C calling convention, i.e., it cannot implement calls to functions taking more arguments on the stack than the caller as proper tail calls. Furthermore, there are a lot of pragmatic restrictions:

- On the Alpha, it can only handle calls to static functions, due to the GP problem (see section 6.8.1).
- When generating position independent code on the i386, it can only handle calls to static functions.
- It cannot handle indirect calls (calls through function pointers).
- Calls to variable argument functions are not handled.
- It does not handle the following case: Outgoing argument a is evaluated before outgoing argument b, but the evaluation of b depends on the value in the stack which is overwritten by a.
- It does not handle calls if the caller stores any local variables or temporaries on the stack.

The optimization operates entirely at the RTL level. The code producing the RTL code for the function call (in `calls.c`) tries to produce two versions of the call: the normal function call sequence and the proper tail call sequence. The normal call sequence is always produced, while the proper tail call sequence generation can fail for several reasons (see above). The reason why two sequences are produced is that it is not known at this stage whether the call is in a tail position. Instead of the actual code for the call, a `call_placeholder` expression containing the generated sequences is inserted into the RTL code.

After generating RTL code for the function, a separate pass over the RTL code (in `sibcall.c`) detects the tail calls, performs a few other tests to determine whether they can be coded as tail calls, and selects the appropriate calling sequences accordingly.

6.5 Requirements for Proper Tail Calls

In section 3.1 we have identified two requirements that must be fulfilled so that a tail call can be implemented as a proper tail call:

- The call must be in a tail position.
- An address of a non-static local variable of the function may only be stored in another non-static local variable of that function.

Section 3.1.1 describes how to solve the first requirement by changing and extending the C grammar. While being a nice solution from a theoretical point of view, this is not very feasible to do in practice, if only for the code duplication and maintenance overhead.

Clean solutions corresponding closely to the grammar extension would be to use an attributed grammar [ASU85] or to annotate the parse tree after parsing. Unfortunately, neither of these are options in GCC. The former is not because GCC uses the Bison parser generator, which does not provide for attributed grammars, and switching to an attribute grammar system like Ox [Bis93] would almost certainly not be approved of by the maintainers. Annotating the parse tree after parsing is not an option either because the whole parse tree is never constructed. Instead, GCC generates parse trees for expressions but produces RTL code for statements immediately, discarding all parse trees after their corresponding RTL code has been generated.

So we are back at extending the grammar. I chose not to do this for the abovementioned reasons and because it, too, would be unlikely to be accepted by the GCC maintainers. Instead, I decided for the simplest solution: use the existing code.

The code in GCC (in `sibcall.c`) that determines whether a call is in a tail position operates at the RTL level. The logic it applies is quite simple: If the sole successor of the block containing the call is the exit block, the call is a tail call.

The existing tail call optimization in GCC has two redundant checks to assure that the second requirement is met: First it makes sure that the calling function does not store any local variables or temporaries on the stack. After RTL generation it also prevents tail calls in functions whose RTL code contains an `ADDRESSOF` expression[1].

Forbidding the `ADDRESSOF` expression in RTL code basically means that a function which uses the C address operator `&` even on a global variable cannot make tail calls. The restriction concerning stack usage means that functions evaluating more complex expressions (the complexity limit depending on the number of available registers, i.e., on the target architecture) cannot make tail calls. These checks are clearly much too restrictive and too unpredictable.

There are two ways an address of a non-static local variable can be stored in some location other than a non-static local variable: by direct assignment or by being passed as a parameter to a function. I have taken a conservative but simple approach to the problem. If the address of a non-static local variable or a value from which it can be reconstructed is found on the right hand side of an assignment or in the parameter list of a function call, the function being compiled will not make proper tail calls.

A less restrictive test would be to prohibit only assignments to non-local and static local variable and of uses as actual parameters, but the following examples makes clear that this requires data flow analysis:

```
void f (void) {
    int x = 0, *y;
    y = &x;
    g(y);
}
```

It would also be possible to differentiate between different tail calls:

```
void f (int a) {
    int x = 0, *y = 0;
    if (a)
        g(y);
    else {
        y = &x;
        g(y);
    }
}
```

Here, the first call to `g` could be implemented as a proper tail call, but not the second. Anyway, I have refrained from implementing more complex

[1] `ADDRESSOF` describes the address of a temporary. Stack slots are allocated for temporaries whose addresses are taken.

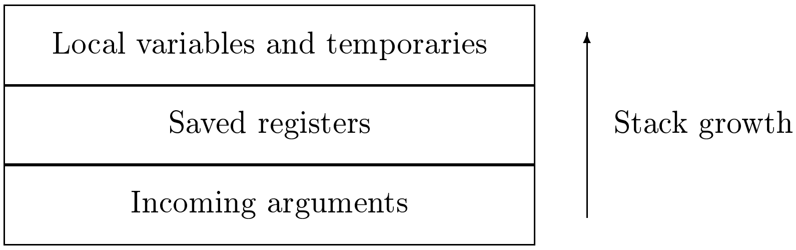

Figure 6.1: Stack frame layout

analyses because the simple test mentioned above is absolutely predictable and the case above can easily be avoided by putting the `else` branch into a separate function.

I have implemented this check to occur during parsing, i.e., while tree code is constructed and translated to RTL code. For assignments and function calls it is checked whether the right hand side respectively the actual parameters contain as a subexpression the address of a non-static local variable. In C, there are two ways to get such an address: use the address operator or implicitly convert an array into a pointer. For example, if `a` is declared as `int a[3][4]`, the expression `a` is the address of the whole array. `a[0]` is the address of the first sub-array and `a[0][0]` is the first element of the first sub-array. Only the latter is not an address. One also has to account for arrays within structs, for example, but overall, the code to perform this test is rather simple.

6.6 Architecture Independent Issues

Figure 6.1 depicts the layout of an activation record. If a proper tail call is to be performed, the outgoing arguments have to be placed in the incoming argument space of the caller, which is at the bottom of the activation record. This poses two problems:

- The incoming argument space might not be large enough to hold all outgoing arguments.
- The values of the outgoing arguments might depend on the incoming arguments.

6.6.1 Space for Outgoing Arguments

The first problem can easily be solved by reserving the additional amount between the incoming arguments and the saved registers. A solution not wasting that space would have to be less efficient in the general case. A simple, but very inefficient strategy would be to copy the part of the space to be overwritten somewhere not affected, before placing the outgoing arguments there. It would be more efficient to evaluate and place the arguments in an order that would not need the copying of any stack slot. For simplicity's sake, let us disregard the saved registers space and pretend that the local variables are above the incoming arguments space. Furthermore, let us assume that every outgoing argument depends on two specific local variables, which, due to their position on the stack, would have to be overwritten. Both of them must be intact when the last outgoing argument is to be evaluated, i.e., at least one outgoing argument must not have been placed at its final position (if it were, one of the local variables would have been overwritten). One could argue that in that case the local variables should have been placed somewhere else. Assume that there were no other local variables or temporaries in the caller besides these two. It is now clear that there needs to be at least one empty stack slot above the incoming arguments. Although in most cases it might be possible to save some stack space in comparison to the simple reservation approach, I decided against it because the cost of implementation is too big and the potential benefit too little.

For variable argument functions, the implementation calculates the additional amount of stack space to be reserved on the assumption that only the fixed arguments are passed to the function (the worst case). This is simple and does not add overhead to the generated code.

6.6.2 Dependencies between Incoming and Outgoing Arguments

The second problem is quite similar to the clash between outgoing arguments and local variables discussed in the previous subsection. Here it is even more obvious that in some cases, either an incoming or an outgoing argument must be copied. A trivial solution would be to evaluate the outgoing arguments but place them somewhere on the stack where they do not overwrite the incoming arguments. I have taken a similar, but more efficient approach: The compiler allocates a temporary for each outgoing argument and places all outgoing arguments in their temporaries. After all of them have been evaluated, the contents of the temporaries are copied to the positions of their corresponding arguments on the stack. That way, GCC will analyze which incoming arguments it can overwrite without sacrificing the correctness of the resulting code. For the other temporaries, it will allocate stack slots.

6.6.3 Stack Pointer Readjustment

On RISC architectures, it is customary that the stack frame allocated by a function includes the space for the outgoing arguments. That way, a function needs to modify the stack pointer only twice: on entry and on exit. If we want to keep that arrangement, we must "unpop" the outgoing argument space that has been popped by a callee after a non-tail call. This seems to be inefficient, but consider the alternative. If the outgoing argument space were not preallocated, we would have to modify the stack pointer before each function call, i.e., at least as often as for the preallocated case. In fact, we actually save a stack pointer modification in the case of tail calls, since they do not return to their direct callers.

6.6.4 Variable Argument Functions

As argued in chapter 5, variable argument functions using the proptail calling convention are passed an implicit argument, namely a pointer to the bottom of the incoming arguments on the stack, which I refer to as the "bottom pointer". This is also the address the stack pointer has to be restored to when the function returns, as well as the address outgoing arguments for a proper tail call have to be placed relative to. I have chosen to pass this address as the first argument without any special-casing in the argument passing convention.

The only serious problem is setting the stack pointer to this address or relative to this address before exiting a function or performing a proper tail call, since this happens in GCC in machine specific code in a pass after register allocation, which means that the register allocator does not know that the argument will be used in the epilogue unless there were some clean way of communicating that fact, which to my knowledge there is not. Hence I took the simple approach of saving the first argument in variable argument proptail functions in the stack frame in the prologue and fetching it from there in the epilogues. The actual implementation is architecture dependent.

One improvement to this simplistic approach on the i386, which passes all arguments on the stack, would be to take the bottom pointer from its incoming argument slot when returning from a function. Then the additional saving of this argument in the prologue could be omitted for variable argument functions which do not perform proper tail calls. Note that this approach cannot be taken for setting the stack pointer before a proper tail call since the stack slot for the bottom pointer might already have been overwritten by the outgoing proper tail call arguments. Note also that this is not a valid approach on RISC architectures, since this argument is passed in a register and may not be live at that point. On the other hand, we know that the bottom pointer in a proper tail call to another variable argument function must contain the same address as the bottom pointer given to the caller, so we could speed up this case (a variable argument function performing a proper tail call to a variable argument function) a bit.

6.7 A Summary of all Scenarios

1. Calling a proptail function from a non-proptail function.

   Such a call is never a proper tail call.

   (a) Calling a non-vararg function

      i. Push arguments
      ii. Call
      iii. Readjust stack pointer

   (b) Calling a vararg function

      The first argument for this kind of call is the bottom pointer.

      i. Push arguments
      ii. Call
      iii. Readjust stack pointer

2. Calling a proptail function from a proptail function

   (a) The call is not a proper tail call

      The calling sequences are the same as for case 1.

   (b) The call is a proper tail call

      i. Calling a non-vararg function

         A. Place the arguments in their tail call positions (the space beginning with the incoming arguments of the callee). If the caller is a vararg function, this is the area beginning at where the bottom pointer points to.
         B. Adjust the stack pointer so that it points to the end of the arguments
         C. Call

ii. Calling a vararg function
The first argument for this kind of call is the bottom argument pointer. If the caller is itself a vararg function, the value for this argument is the same as the caller got.
    A. Place arguments in their tail call positions
    B. Adjust the stack pointer so that it points to the end of the arguments
    C. Call

6.8 Architecture Dependent Issues

6.8.1 Alpha

Variable Argument Functions

Section 6.6.4 describes the problems and the solution with accessing the bottom pointer in the epilogues in variable argument functions. The specific solution on the Alpha is to save the first argument register (`$16`) together with the callee-saved registers on the stack.

In the epilogues, this address is loaded as the last thing before the jump. In the epilogue before a proper tail call, this address must additionally be offset by the size of the outgoing arguments, which takes an additional instruction. See section 6.9.2 for an example.

GP

Section 4.1.1 elaborates on the use of the register `$gp` for addressing static constants. It is set to a compilation unit specific value upon entry to a function. This has the consequence that after a call to a function which could set the register to another value, the register must be set again. Since that value is the same for all functions in the same compilation unit, that had to be done after calls to functions in other compilation units. Proper tail calls change that, however. Assume that `f` and `g` are functions in the same compilation unit, and that `h` is a function in another compilation unit. `f` performs a call to `g`, which performs a proper tail call to `h`. `h` returns directly to `f`, so that the value in `$gp` is not the right value for `f`, although `f` did a normal call to a function in its own compilation unit. The existing tail call optimization (see section 6.4) makes this scenario impossible by forbidding proper tail calls to non-static functions. Unfortunately, we do not have this luxury. The solution is to set `$gp` after each call to a proptail function.

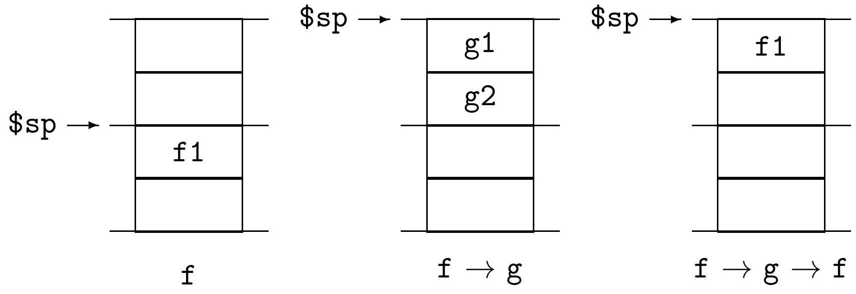

Figure 6.2: The problem with stack alignment on the Alpha

Stack Alignment

Section 4.1.1 mentions that the stack pointer must always be aligned on a 16 byte boundary. This has a subtle implication on the calling convention for proper tail calls. Imagine a function `f` taking one argument (8 bytes) on the stack and a function `g` taking two arguments (16 bytes) on the stack. `f` and `g` are mutually properly tail recursive. Figure 6.2 illustrates this example. The vertical separators denote 16 byte boundaries. Assume `f` is passed its stack argument at location p on the stack, which means that `$sp` points to p upon entry to `f`. Clearly, p must be 16 byte aligned. `f` must now place the outgoing arguments on the stack. The obvious locations, namely $p - 8$ and p cannot be used, since $p - 8$ is not 16 byte aligned and the first stack argument must always be passed at an address so aligned. Hence, if `f` cannot assume that the 8 bytes at $p + 8$ are not used by the caller of `f`, it must place the arguments at $p - 16$ and $p - 8$. `g` now places its stack argument for `f` at location $p - 16$ and calls `f` with `$sp` pointing there. This is a situation where two mutually recursive functions use unbounded stack space, i.e., they are not properly tail recursive.

The conclusion is that the proptail calling convention for the Alpha must include the constraint that code calling a proptail function with an even number n of stack arguments must assume that the value in the position where the hypothetical stack argument $n + 1$ would be placed (the location 8 bytes higher than the location of the last stack argument) is clobbered by the call.

The Alpha calling convention manual [Com00b] is not clear on this topic (it does not even mention it), even though GCC always produces code obeying that constraint.

6.8.2 i386

Variable Argument Functions

The bottom pointer is saved between the return address (plus additional space for outgoing proper tail call arguments if the needed size for outgoing proper tail call arguments is larger than the size of the incoming arguments) and the saved frame pointer (which is kept in register `%ebp`) or, if the frame pointer has been omitted (via the `-fomit-frame-pointer` option), the saved callee-saved registers.

The epilogue for proper tail calls is straightforward: After restoring callee-saved registers, pop the saved bottom pointer into the stack pointer register `%esp`, subtract the size of the outgoing arguments, and jump.

The epilogue for function exits is a little more complicated because of a bug in the version of GCC I used[2]: A function exit epilogue may not use a `jmp` instruction to return. Hence, it must be arranged for the return address to be on top of the stack after setting the stack pointer to the bottom pointer. This address must therefore be moved to a register first and, after the stack pointer has been modified, pushed. Then, a `ret` will have the desired effect. On modern i386 implementations, this seemingly awkward maneuver might actually improve performance, since these processors have a return address stack, which speeds up `ret` instructions. Using a `jmp` instead of a `ret` would completely invalidate this cache, leading to costly misses. See chapter 8 for another approach to this problem.

See section 6.9.1 for an example of both epilogues.

Position Independent Code

In position independent code on the i386, global data is referenced through a global offset table, whose address is determined upon function entry relative to the program counter and stored into the register `%ebx`. Since `%ebx` is a callee-saved register, it must be restored before executing a proper tail call. This means that the address of the function to call must be loaded from the global offset table before restoring `%ebx` and must be saved into a caller-saved register. This register can then be used as the target of the proper tail call.

6.8.3 SPARC

Although I have not yet implemented support for the proptail calling convention on the SPARC, I want to discuss the problems this architecture poses,

[2]The bug has been fixed in a later CVS version, but I have not upgraded yet.

`%sp` →	64 bytes for the system to save register window
	Pointer to structure return value
	24 bytes to store register arguments
	Incoming arguments past the sixth, if any

Figure 6.3: SPARC stack contents upon function entry

since it differs significantly from the other two architectures and it is not obvious that the calling convention modifications detailed in chapter 5 are even possible on the SPARC.

The biggest obstacle is that the stack pointers of all register windows used by user code must always point to 64 byte regions of memory not modified by user code. Furthermore, it is forbidden to modify the stack pointer of any register window other than the current (the only such register window would be the next higher one, the stack pointer of which is the frame pointer of the current register window), since the system might have “swapped out” that register window and modifying the contents of its save region would mean that its registers contents would be clobbered when it is restored again. It is not even possible to copy this region and modify the stack pointer of that window to point to the copy since a trap (through an interrupt, for example) may occur between any two instruction. In other words: We may never modify the frame pointer, neither may we modify the 64 bytes it points to.

It is not forbidden, however, to modify the stack pointer of the current register window, as long as we make sure that we do not use the 64 bytes it points to. Figure 6.3 illustrates the contents of the stack upon function entry (before executing a potential `save` instruction).

The first thing to do is to enlarge the stack frame so that it can hold outgoing arguments to all proper tail calls the function makes. This means that the stack pointer might have to be modified. This modification automatically means that the 64 byte region for the system to save the register window moves accordingly. The potential pointer to the memory block where a structure return value must be copied to does not need to be moved now, neither does the region for storing register arguments. After having modified the stack pointer, the `save` instruction may be issued.

The next question is how to exactly handle a proper tail call. First, the callee must be entered with the same register window current as was current upon entry to the caller. Otherwise, a proper tail call would consume at least 64 bytes of memory to save a register window. This means that the outgoing register arguments must be stored in registers `%i0`–`%i5` instead of `%o0`–`%o5`, at least if the caller uses `save`/`restore`. Outgoing stack arguments must be written over the incoming stack arguments, as usual. Here, care must be taken that the pointer to the memory block for a structure return value be moved to the position it must occupy upon entry to the callee before it is overwritten by an outgoing argument, which happens if the callee takes at least 7 more stack arguments than the caller, and may happen if the caller is a variable argument function and the callee takes at least 7 stack arguments.

After this has been done, the caller must issue the `restore` instruction if it used the `save` instruction in the prologue. The only thing left to be done now before executing the call is to modify the stack pointer to point exactly 92 bytes above the highest outgoing argument.

In the proptail calling convention, the callee must pop its arguments from the stack. It is obvious that we cannot just set the stack pointer to the point below the incoming arguments, since that space is likely to be used by the caller for temporaries, which might be destroyed if a trap occurred. The simple solution is to simply set the stack pointer to 64 bytes above the end of the incoming arguments, so that a possible trap is taken care of. It is then up to the caller to readjust the stack pointer to where it was before the call, just as it must be done on other architectures as well.

Since the stack pointer on the SPARC must be aligned on an 8 byte boundary, but arguments only on 4 byte boundaries, the same minor stack alignment problem occurs as on the Alpha. See section 6.8.1 for the solution.

The standard SPARC calling convention specifies that structures are passed to the callee by reference. The proper tail call implementation presented in this work assumes that tail calls passing structures must be implemented as proper tail calls if the requirements are met (see section 6.5). If that is to be upheld, the SPARC proptail convention will have to pass structures by value, like on the Alpha and the i386. Whether or not that should be done I leave open, but it has to be considered and documented before the proptail code is included in the GCC distribution.

6.9 Generated Code

Now we take a look at the code generated by the modified GCC. We will use two examples to demonstrate most of the phenomena described in this chapter. The first one consists of two simple, mutually recursive functions:

```
int fa1 (int x, int i, int n, int dummy) {
    if (i > n)
```

```
        return x;
    return fa2(x * i, i + 1, n);
}

int fa2 (int x, int i, int n) {
    if (i > n)
        return x;
    return fa1(x * i, i + 1, n, 0);
}
```

It is not hard to see that they are identical except for the `dummy` argument of `fa1`, which I introduced so that the two functions have a different number of incoming arguments.

The second example is a directly recursive function with variable arguments, which computes the same mathematical function as the two functions above, namely the factorial:

```
int fav (int x, int i, int n, ...) {
    if (i > n)
        return x;
    return fav(x * i, i + 1, n, 1, 2, 3, 4);
}
```

6.9.1 i386

Let us first examine code generated for the i386. The function `fa1` is compiled to this assembler code using the options `-O3 -fomit-frame-pointer -mpreferred-stack-boundary=2 -fno-inline -fno-schedule-insns -fno-schedule-insns2`. Instruction scheduling has been omitted to make the generated code more readable:

```
fa1:
        pushl   %esi
        pushl   %ebx
        movl    12(%esp), %ebx
        movl    16(%esp), %ecx
        movl    20(%esp), %esi
        cmpl    %esi, %ecx
        jle     .L40
        movl    %ebx, %eax
        popl    %ebx
```

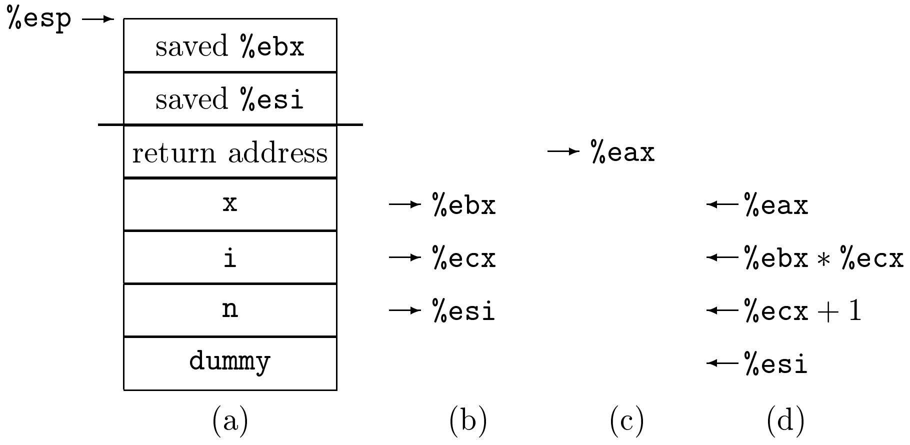

Figure 6.4: Stack contents during the execution of `fa1`

```
        popl    %esi
        ret     $16
.L40:
        movl    8(%esp), %eax
        leal    1(%ecx), %edx
        imull   %ecx, %ebx
        movl    %eax, 12(%esp)
        movl    %esi, 24(%esp)
        movl    %edx, 20(%esp)
        movl    %ebx, 16(%esp)
        popl    %ebx
        popl    %esi
        popl    %ecx
        jmp     fa2
```

The contents of the stack during the stages of the execution of `fa1` are illustrated in figure 6.4. Figure 6.4a shows the contents of the stack and of `%esp` after the two registers `%ebx` and `%esi` have been pushed. The vertical rule marks the end of the incoming argument space. Figure 6.4b shows which stack slots are moved into which registers. This is done directly after the two pushes.

The code for the consequence of the `if` statement (before the label `.L36`) restores not only the two saved registers but, by virtue of the argument to the `ret` instruction, also pops the incoming arguments.

%esp →	return address
	x ∗ i
	i + 1
	n

Figure 6.5: Stack contents for `fa1` before the jump to `fa2`

The code after the `if` statement is much more interesting, as it performs a proper tail call. The function called is `fa2`, which takes one argument less than `fa1`. The result of this is that `fa1` has to modify the stack accordingly, which includes moving the return address to a lower position. Figure 6.4c illustrates that the return address is moved into the register `%eax`.

Figure 6.4d shows the values which are then written onto the stack over the incoming arguments. After that, `%ebx` and `%esi` are restored and the return address is popped off the stack[3].

Figure 6.5 shows the result of these stack manipulations, i.e., the contents of the stack before the jump to `fa2`. Note that the return address on the stack is the same address that `fa1` had to return to, i.e., it does not lie within `fa1`.

This is the code for `fa2`, compiled with the same options:

```
fa2:
        subl    $4, %esp
        pushl   %esi
        pushl   %ebx
        movl    16(%esp), %ebx
        movl    20(%esp), %ecx
        movl    24(%esp), %esi
        cmpl    %esi, %ecx
        jle     .L42
        movl    %ebx, %eax
        popl    %ebx
        popl    %esi
```

[3]The compiler uses `pop %ecx` because this is more compact (`%ecx` is a caller-saved register, so it can be modified at will by `fa1` without saving it (see section 4.1.2)). The more obvious instruction would have been `addl $4, %esp`.

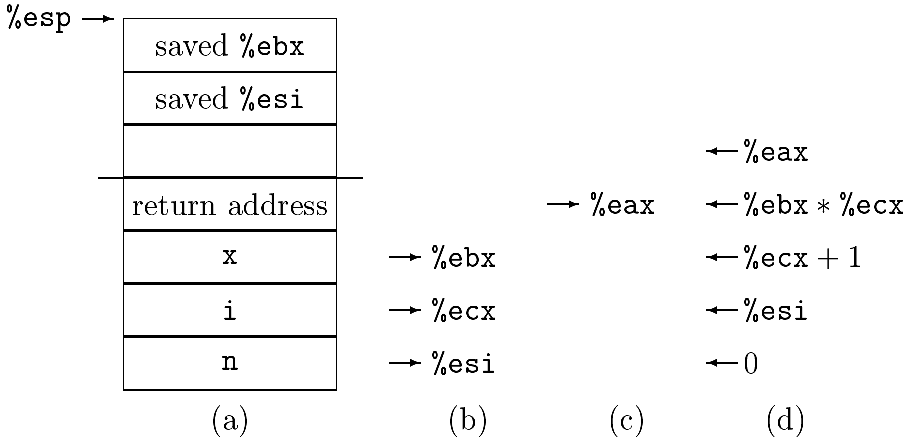

Figure 6.6: Stack contents during the execution of `fa2`

```
        popl    %edx
        ret     $12
.L42:
        movl    12(%esp), %eax
        leal    1(%ecx), %edx
        imull   %ecx, %ebx
        movl    %eax, 8(%esp)
        movl    $0, 24(%esp)
        movl    %esi, 20(%esp)
        movl    %edx, 16(%esp)
        movl    %ebx, 12(%esp)
        popl    %ebx
        popl    %esi
        jmp     fa1
```

Figure 6.6a shows the contents of the stack and the stack pointer after `%esi` and `%ebx` have been saved. Note that the compiler generated code to reserve one stack slot before pushing these two, which corresponds to the empty box above the vertical rule, which again marks the end of the incoming argument space. The reason for this empty slot is that `fa2` calls `fa1`, which takes one argument more than `fa2`, hence the incoming argument space alone is not sufficient to hold all arguments. Figure 6.6b show which registers the incoming arguments are saved in. The code for the consequence of the `if` statement not only has to restore the two saved registers but needs to pop the empty stack slot and the incoming arguments as well.

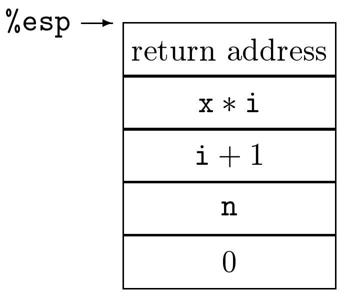

Figure 6.7: Stack contents for `fa2` before the jump to `fa1`

The code for the proper tail call again has to reorganize the stack. In this case, the return address, which is moved into `%eax` (illustrated in figure 6.6c) has to be moved up one stack slot, i.e., into the empty stack slot. This and the other stores into the stack are depicted in figure 6.6. That being done, the two saved registers have to be restored and `fa1` must be jumped to. The contents of the stack and the stack pointer are shown in figure 6.7.

The following is the code for the variable argument function `fav`:

```
fav:
        subl    $16, %esp
        movl    20(%esp), %edx
        pushl   %edx
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        movl    36(%esp), %ecx
        movl    40(%esp), %esi
        movl    44(%esp), %ebx
        movl    48(%esp), %edi
        cmpl    %edi, %ebx
        jle     .L46
        movl    %esi, %eax
        popl    %ebx
        popl    %esi
        popl    %edi
        movl    20(%esp), %edx
        popl    %esp
        pushl   %edx
```

```
            ret
22    .L46:
            movl    32(%esp), %eax
24          leal    1(%ebx), %edx
            imull   %ebx, %esi
26          movl    %eax, -36(%ecx)
            movl    $4, -4(%ecx)
28          movl    $3, -8(%ecx)
            movl    $2, -12(%ecx)
30          movl    $1, -16(%ecx)
            movl    %edi, -20(%ecx)
32          movl    %edx, -24(%ecx)
            movl    %esi, -28(%ecx)
34          movl    %ecx, -32(%ecx)
            popl    %ebx
36          popl    %esi
            popl    %edi
38          popl    %esp
            subl    $36, %esp
40          jmp     fav
```

Line 2 reserves 4 stack slots which may be used for the outgoing proper tail call arguments. `fav` takes at least three arguments, but makes a proper tail call with 7 arguments, so in the worst case (it only gets 3 arguments) 4 more stack slots are required.

Line 3 saves the bottom pointer argument to the stack.

Lines 15 to 21, excluding line 15 are the epilogue of the function exit. First, the callee-saved registers are restored. Then, in line 18, the return address is moved to register `%edx`. Line 19 sets the stack pointer to the bottom pointer and lines 20 and 21 jump to the return address (see section 6.8.2 for why this is not implemented with a `jmp` instruction).

Lines 35 to 39 are the epilogue for the proper tail call. After restoring the callee-saved registers, the stack pointer is set to the bottom pointer minus 36, which is the space needed for the seven explicit arguments, the bottom pointer argument and the return address. The indirect jump in line 40 finally commits the proper tail call.

Figure 6.8a shows the contents of the stack after line 7 has been executed. It also shows where the bottom pointer points to. The space at the bottom with the three dots depicts the arguments coming after the three mandatory arguments. It occupies two stack slots, but in general, it might be arbitrarily large or small, including length zero. Figure 6.8b shows which stack slots are

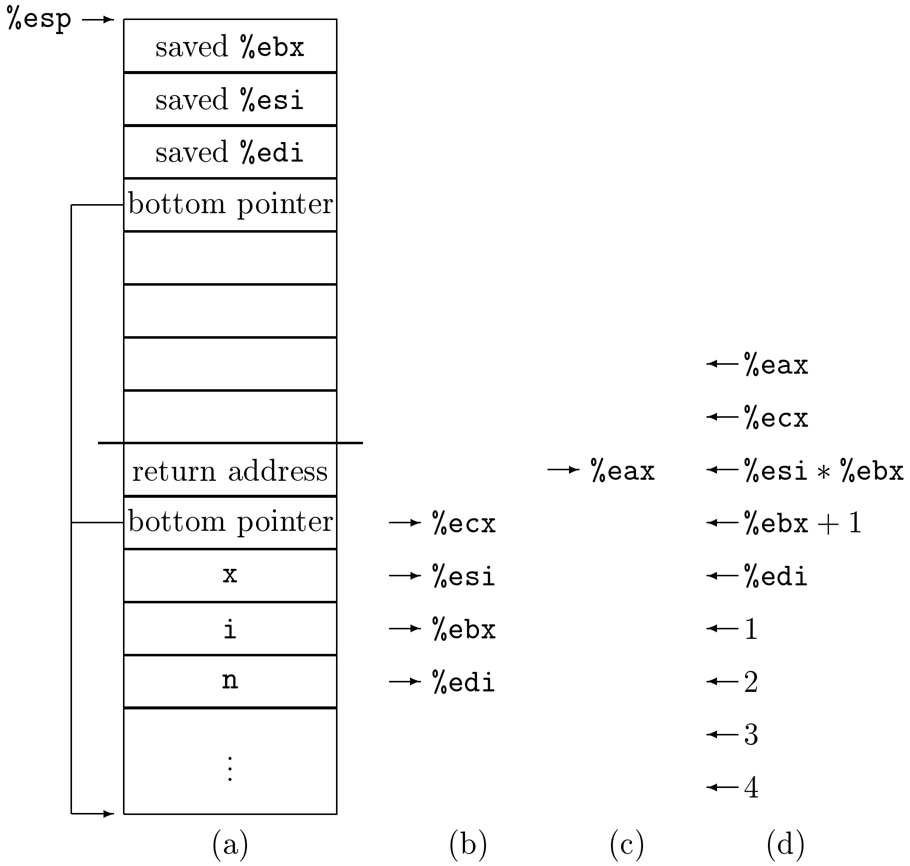

Figure 6.8: Stack contents during the execution of `fav`

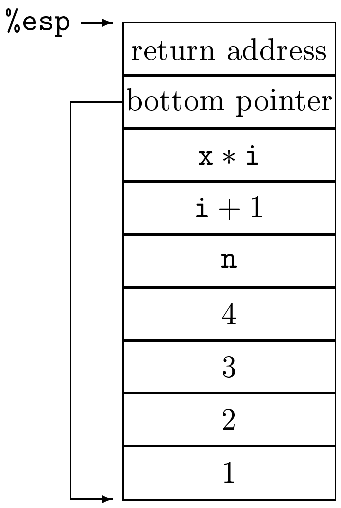

Figure 6.9: Stack contents for `fav` before the proper tail call

copied to which registers in lines 8 to 11. Figures 6.8c and 6.8d show the stack operations for the proper tail call. The former illustrates the copying of the return address to `%eax` in line 23, and the latter the building of the new stack frame in lines 26 to 34. Here we see the reason for the four empty stack slots: They guard against the case of the new stack slot overwriting parts of the old stack slot which are still needed, in this case the saved registers and the copy of the bottom pointer. Should the new stack frame contain too many elements for the register set to hold, some values would have to be spilled. The stack slots holding these values would be above the saved registers, i.e., could never be accidentally overwritten by the new stack frame.

Figure 6.9 finally shows the contents of the new stack frame right before the jump to `fav` in line 40. It also illustrates that the bottom pointer points to the bottom of the outgoing arguments. Note that this is the same location as it pointed to upon entry to the function, as shown in figure 6.8.

6.9.2 Alpha

The situation on the Alpha is quite a bit different. Here, the first six arguments plus the return address are passed in registers.

The code for the Alpha has been generated using the options `-O3 -fno-inline -fno-schedule-insns -fno-schedule-insns2`. Again, I have omitted instruction scheduling so that the code be more readable. Let us look at the generated code for `fa1`:

```
fa1:
        ldgp $29,0($27)
$fa1..ng:
        lda $30,-16($30)
        stq $26,0($30)
        cmple $17,$18,$1
        mov $16,$0
        beq $1,$L35
        mull $16,$17,$16
        addl $17,1,$17
        ldq $26,0($30)
        lda $30,16($30)
        jmp $31,fa2
$L35:
        ldq $26,0($30)
        lda $30,16($30)
        ret $31,($26),1
```

Surprisingly, GCC generates code which allocates a stack frame and even saves the return address there. The reason for this is that this happens in the target machine specific code for the Alpha and I have not taken the time to optimize this in the case where a function only does proper tail calls. The potential benefits of such an optimization are analyzed in chapter 7, though. Note that the allocated stack frame is 16 bytes large, although only 8 bytes are used. This is due to the 16 byte stack alignment constraint (see section 6.8.1).

Apart from the saving and restoring of the return address register (`$26`) and the stack frame allocation thus necessitated, there is nothing surprising here. The `dummy` argument, passed in register `$19`, is simply ignored.

The code for `fa2` is very similar:

```
fa2:
        ldgp $29,0($27)
$fa2..ng:
        lda $30,-16($30)
        stq $26,0($30)
        cmple $17,$18,$1
        mov $16,$0
        beq $1,$L37
```

```
        mull $16,$17,$16
        addl $17,1,$17
        mov $31,$19
        ldq $26,0($30)
        lda $30,16($30)
        br $31,$fa1..ng
$L37:
        ldq $26,0($30)
        lda $30,16($30)
        ret $31,($26),1
```

There is one obvious difference between these two functions. `fa1` jumps to the label `fa2` whereas `fa2` jumps directly to `fa1..ng`, avoiding the reload of the `$gp` register (see section 4.1.1). The two functions were obviously compiled in the same compilation unit, because the latter jump could not have legally been shortcut otherwise. Since GCC compiles functions one by one, there was no way for it to know at the time of the compilation of `fa1` that `fa2` would be contained in the same compilation unit, so it had to conservatively assume otherwise. The linker should detect such cases and optimize the jumps, although the GNU linker does not do so.

Let us now look at variable argument functions, as they require some special handling on RISC machines due to the passing of arguments in registers. The code produced for the function `fav`, using the same compiler options as above, is this:

```
fav:
        ldgp $29,0($27)
$fav..ng:
        lda $30,-144($30)
        stq $26,16($30)
        stq $16,24($30)
        mov $16,$3
        stq $19,120($30)
        stq $20,128($30)
        stq $21,136($30)
        stt $f19,72($30)
        stt $f20,80($30)
        stt $f21,88($30)
        cmple $18,$19,$1
        mov $17,$0
        beq $1,$L39
```

```
            lda $1,3
18          lda $2,4
            mull $18,$17,$17
20          addl $18,1,$18
            lda $20,1
22          lda $21,2
            stq $1,-16($3)
24          stq $2,-8($3)
            ldq $26,16($30)
26          ldq $22,24($30)
            lda $30,-16($22)
28          br $31,$fav..ng
    $L39:
30          ldq $26,16($30)
            ldq $30,24($30)
32          ret $31,($26),1
```

Before we examine the inefficiencies of this code, let us look at the proper tail call specific things it does.

Since it is the first argument, it is stored in the register `$16`. In line 6 it is stored on the stack, in line 7 copied to register `$3`. In lines 23 and 24 the two outgoing arguments to be stored on the stack are stored in locations relative to this address. Furthermore, the stack pointer is set relative to this address before performing the proper tail call in line 27 (after fetching it from the stack in the preceding line). Before leaving the function, the stack pointer is set to this address in line 31. Note also that this bottom pointer is passed through to the callee, which is also a variable argument function, directly. This always happens at proper tail calls between variable argument functions.

The inefficiencies in this code are quite numerous:

- Integer and floating point argument registers beginning with the fourth are saved to the stack, but never restored.
- The stack frame is unnecessarily large.
- The bottom pointer argument is copied to the stack as well as to register `$3`, all the while residing unchanged in register `$16`.

The reason for the first two inefficiencies lies in GCC's implementation of variable arguments on the Alpha. The `va_arg` macro, which accesses arguments past the last mandatory argument, operates solely on the stack, so all registers which could contain variable arguments are saved to the stack. The reason why the stack frame is so large is that GCC always allocates enough space to save all argument registers, even if that much space is not necessary, probably to keep the implementation simple. There are 12 argument registers (`$16`–`$21` and `$f16`–`$f21`), each 8 bytes wide, so this accounts for 96 bytes. The rest is reserved for saving other registers and for the outgoing arguments for the proper tail call.

See section 6.8.1 for why the first argument is copied to the stack.

6.10 Limitations

The current implementation has some minor limitations:

- It never implements calls from a function using `alloca` as proper tail calls. This limitation is no more restrictive in practice than the constraint that no address within the stack frame may be assigned to a variable or passed as a parameter, since not assigning the result from `alloca` as a variable or passing it as a parameter is a rather useless exercise.

- It does not implement calls from a function using variable size arrays, which are a GCC extension as well as part of the C99 standard [Ame99]. The reason is simply that implementing that would require additional effort and I see the support of this feature as being of minor importance.

- Variable argument arrays are not supported as arguments to proptail functions. Again, the reason is that this does not have a priority for me.

- Position independent code is not yet supported on the i386. Section 6.8.2 outlines the problem with proper tail calls and PIC on the i386 and the solution to that problem. It will be implemented soon.

Chapter 7

Performance

In this chapter the run-time performances of the proptail calling convention and the standard C calling convention (without the existing tail call optimization described in section 6.4), both as implemented in GCC, are compared. The numbers in this chapter have to be taken with a grain of salt, firstly because one should never take benchmarks too seriously and secondly because an unnumbered CVS version of GCC was used for all these tests. Furthermore, it must be stressed that the proptail convention implementation has not been optimized for speed yet and, as will be shown, there are some areas where performance can be greatly improved. Chapter 8 outlines some optimizations which I may pursue the implementation of in the near future. The goal of this chapter is simply to show that the proptail calling convention provides performance comparable to that of the standard calling convention and may, if optimized, provide much superior performance in certain cases on RISC architectures to the standard calling convention in the future.

Times are always given in seconds. They are the arithmetic means of the user times of three runs on an unloaded system. System time was negligible in all runs and the user time differences between the runs were always little enough to be caused by the system clock granularity.

The systems used for the tests are a 500 MHz 21264 Alpha and a 300 MHz Pentium II.

7.1 Fixed Argument Functions

The first benchmark compares the performance of very simple function calls. Given the two functions

```
int fa1 (int x, int i, int n, int dummy) {
    if (i > n)
```

Compiler flags	Standard	Proptail
`-O3 -fno-inline`	10.46	12.72
`-O3 -fno-inline -mpreferred-stack-boundary=2`	9.20	12.68
`-O3 -fno-inline -fomit-frame-pointer`	9.16	11.83
`-O3 -fno-inline -mpreferred-stack-boundary=2 -fomit-frame-pointer`	7.31	10.32

Table 7.1: Run times with varying compiler flags on the Pentium II 300

```
        return x;
    return fa2(x + i, i + 1, n);
}

int fa2 (int x, int i, int n) {
    if (i > n)
        return x;
    return fa1(x + i, i + 1, n, 0);
}
```

I measured the time taken for the loop

```
for (i = 0; i < 10000000; ++i)
    fa1(1, 1, 20, 0);
```

The options that were used for compiling are `-O3 -fno-inline`. Additionally, on the i386 the options `-mpreferred-stack-boundary=2 -fomit-frame-pointer` were used, because they give better performance. Table 7.1 shows the run times for the loop on the i386 with and without the proptail convention with various compiler flag combinations. All other results for the i386 presented here exhibit similar behavior with regard to compiler flags, so I refrain from presenting the other (slower) run times.

On the Alpha, instruction scheduling seems to make code slower sometimes. Table 7.2 presents the run times for the loop on the Alpha with and without proptail convention and instruction scheduling. All following results for the Alpha are obtained by measuring both with and without instruction scheduling and using the better of the two times.

There are two primary factors which can cause the differences in performance between the two calling conventions in this benchmark (apart from instruction scheduling and register allocation):

- Stack modifications. On the i386, the proptail code must move the return address on the stack because the two functions take different

Instruction scheduling?	Standard	Proptail
Yes	3.35	2.70
No	3.36	2.58

Table 7.2: Run times with and without instruction scheduling on the 21264 500

numbers of arguments. This is not an issue on the Alpha, however, because there the return address is passed in a register.

- The number of times a function returns. Since all calls between functions `fa1` and `fa2` are proper tail calls in the proptail convention, the complete execution of the call `fa1(1, 1, 20, 0)` entails only one function return, whereas in the standard calling convention, it entails 21 returns.

The first factor runs in favor of the standard calling convention, while the second is in favor of the proptail convention. On the i386, the first factor is obviously more costly than the second one can make up for. See chapter 8 for how this problem might be solved.

On the Alpha, only the second factor applies for this benchmark, so it may rightfully be assumed that the proptail convention is faster, at least if mostly proper tail calls are performed.

To verify that the advantage is still on proptail's side on the Alpha when stack arguments are involved (which, according to the above reasoning it should be), the same benchmark has been run, but with four dummy arguments added to each function, making `fa1` take 2 stack arguments and `fa2` take one. The proptail version took 2.81 seconds to run and the standard convention version 3.95 seconds.

To demonstrate that proper tail calls exhibit much better performance than standard function calls for deep recursions, due to the improved locality of reference on the stack, the following loop with the same two functions as above has been timed:

```
for (i = 0; i < 1000; ++i)
    fa1(1, 1, 200000, 0);
```

The results are shown in table 7.3.

This is the code generated for `fa1` with instruction scheduling enabled:

Machine	Standard	Proptail
21264 500	13.23	1.81
Pentium II 300	23.48	8.1

Table 7.3: Run times for deep recursion

```
1   fa1:
2           ldgp $29,0($27)
    $fa1..ng:
4           mov $17,$1
            mov $16,$0
6           lda $30,-16($30)
            addl $1,$0,$16
8           addl $1,1,$17
            cmple $1,$18,$1
10          stq $26,0($30)
            beq $1,$L49
12          ldq $26,0($30)
            lda $30,16($30)
14          jmp $31,fa2
    $L49:
16          ldq $26,0($30)
            lda $30,16($30)
18          ret $31,($26),1
```

As can easily be seen, the prologue and the epilogues in this function do unnecessary work. Line 6 reserves a stack frame and line 10 saves the return address. The reverse operations, namely the restoring of the return address, even though the register holding it is not modified, and the freeing of the stack frame are carried out in lines 12, 13 and in lines 16, 17. If these 6 lines are removed, the code runs in 1.20 seconds, as compared to the 1.81 seconds before[1]. This is an optimization which could easily be done by the compiler and may give great performance improvements for code which only performs tail calls, like functional code after CPS transformation [Ste78, App92].

[1]It should be remarked that the 1.20 seconds were achieved with instruction scheduling disabled. The code with instruction scheduling enabled took 1.40 seconds.

Machine	Standard	Proptail
21264 500	4.42	4.99
Pentium II 300	7.87	16.61

Table 7.4: Run times for variable argument functions

7.2 Variable Argument Functions

Variable argument functions are handled differently from fixed argument functions in the proptail calling convention in that they take an additional argument, called the bottom pointer (see chapter 5). To measure the overhead incurred by this convention in contrast to the standard C calling convention, the following two functions are used:

```
int fav1 (int x, int i, int n, int dummy, ...) {
    if (i > n)
        return x;
    return fav2(x + i, i + 1, n);
}

int fav2 (int x, int i, int n, ...) {
    if (i > n)
        return x;
    return fav1(x + i, i + 1, n, 0);
}
```

This loop was timed:

```
for (i = 0; i < 10000000; ++i)
    fav1(1, 1, 20, 0);
```

The only difference between these functions and the ones used for the fixed argument test is that these may be called with arbitrary additional arguments. The results of this benchmark are summarized in table 7.4.

While the speed difference for the Alpha seems to be plausible, this is much less so for the i386. The speed difference is only an indirect consequence of the proptail calling convention. In order to substantiate this claim, let us look at the code generated for the `fav1` function with the proptail convention:

```
fav1:
        movl    4(%esp), %edx
        pushl   %edx
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        movl    28(%esp), %ecx
        movl    32(%esp), %esi
        movl    20(%esp), %ebx
        movl    24(%esp), %edi
        cmpl    %esi, %ecx
        jle     .L84
        popl    %ebx
        movl    %edi, %eax
        popl    %esi
        popl    %edi
        movl    4(%esp), %edx
        popl    %esp
        pushl   %edx
        ret
.L84:
        movl    16(%esp), %eax
        leal    1(%ecx), %edx
        leal    (%ecx,%edi), %ecx
        movl    %eax, -20(%ebx)
        movl    %esi, -4(%ebx)
        movl    %edx, -8(%ebx)
        movl    %ecx, -12(%ebx)
        movl    %ebx, -16(%ebx)
        popl    %ebx
        popl    %esi
        popl    %edi
        popl    %esp
        subl    $20, %esp
        jmp     fav2
```

This is in stark contrast to the code using the standard calling convention:

```
fav1:
        movl    8(%esp), %edx
        movl    12(%esp), %eax
```

```
        movl    4(%esp), %ecx
        cmpl    %eax, %edx
        jle     .L88
        movl    %ecx, %eax
        ret
.L88:
        pushl   %eax
        leal    1(%edx), %eax
        pushl   %eax
        leal    (%edx,%ecx), %eax
        pushl   %eax
        call    fav2
        addl    $12, %esp
        ret
```

The differences between these two versions are twofold:

- The code with the proptail convention has to rewrite the stack, which not only means that it has to cope with an additional argument, namely the bottom pointer, but also with the return address, which it must relocate. This is a direct effect of the proptail convention.
- Having to reorganize the stack results in the three callee-saved registers to be used, which must thus be saved in the prologue and restored in the epilogues. This is an indirect effect of the proptail convention.

The code generated for `fav2` shows similar differences between the two calling conventions.

In order to measure to what amount the register saving and restoring contributes to run-time, the two functions in their standard calling convention versions have been modified to save and restore the three callee-saved registers, even though they are not used in the bodies. This is the code of the modified `fav1` function:

```
fav1:
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        movl    20(%esp), %edx
        movl    24(%esp), %eax
        movl    16(%esp), %ecx
        cmpl    %eax, %edx
```

```
        jle     .L88
        movl    %ecx, %eax
        popl    %ebx
        popl    %esi
        popl    %edi
        ret
.L88:
        pushl   %eax
        leal    1(%edx), %eax
        pushl   %eax
        leal    (%edx,%ecx), %eax
        pushl   %eax
        call    fav2
        addl    $12, %esp
        popl    %ebx
        popl    %esi
        popl    %edi
        ret
```

The run-time of the loop with the modified functions is 13.27 seconds, meaning that the stack reorganizing dictated by the requirements of a proper tail call accounts for a difference of about 3.2 seconds, which is quite plausible when compared to the run-times on the Alpha, given that the Alpha proptail version needs not relocate the return address, since it is passed in a register, and saves two argument registers (see section 6.9.2 for why some argument registers are saved on the stack in variable argument functions) less on the stack than the standard calling convention version, since it takes one argument more (the bottom pointer).

Chapter 8

Conclusion and Further Work

In the preceding chapters I have motivated the need for a way to produce properly tail recursive C code and have described the obstacles such an undertaking poses. I have described a calling convention which allows proper tail calls in C and have presented an implementation of this calling convention for the Alpha and the i386 architectures. I have shown that this calling convention provides acceptable performance and has the potential of outperforming the standard C calling convention under some circumstances if properly optimized.

In this chapter I want to outline some improvements to the implementation, sketch some features which would be useful in conjunction with proper tail calls and describe an implementation technique for functional languages made possible by proper tail calls.

One optimization exemplified in chapter 7 is not to allocate stack frames for function which only make tail calls. On RISC architectures, this would make such tails calls as cheap as jumps.

In order to overcome the performance penalty proper tail calls have on the i386 compared to calls using the standard convention, it might be worthwhile to consider changing the proptail convention on the i386 a bit. By reserving a fixed minimum amount of space for outgoing arguments on the stack and placing the return address at the top of that space, the copying of the return address is made unnecessary for all proper tail calls from functions taking at most that amount of incoming arguments to functions with the same property. The amount of minimum argument space actually used should be determined by empirical tests. Clearly, the larger that amount is, the less cases there are in which the return address must be copied, but the more stack space is used.

Another approach to solve this problem might work if `call` and `ret` are abandoned altogether: Instead of passing the return address as an implicit first argument (aided by the `call` instruction), passing it as an implicit last argument makes moving it around on the stack unnecessary for proper tail call chains. The `call` would have to be split up into `push` and `jmp` instructions for this to work. The problem is that this would cause the processor-internal return address stack to be invalidated if function return is implemented with the `ret` instruction, since return address stack consistency is only guaranteed if `call`s and `ret`s match exactly. Obviously, this can be prevented by implementing function return with `jmp`. Whether this really improved performance has to be tested.

[Bak95] describes a technique for compiling functional code to C by emitting CPS transformed code. This code has the property that all function calls are tail calls. In addition, this technique allocates storage on the C stack and, upon stack overflow, does a precise, copying garbage collection of the C stack. Since no function ever returns, the garbage collector can be ignorant about stack layout. It only needs to know all roots and the layout of the data objects.

GCC already implements a function attribute called `noreturn`. It signifies that a function cannot return and GCC can optimize the caller of such a function without having to regard the consequences of the function ever returning. Unfortunately, it does not optimize the function itself. Obviously, since the function can never return, there is no need to save either callee-saved registers or the return address. Furthermore, local variables would have to be preserved for the last call in the function only if their address might have been taken (see section 6.5). These optimizations would not only speed the code generated by Baker's compilation technique up directly, but would also slow down stack growth, resulting in further speedup due to less frequent garbage collection.

By changing Baker's technique slightly using the proptail convention, we arrive at the C equivalent of what some native code compilers for functional languages, like SML/NJ [App92], do. Using a heap instead of the C stack to allocate data structures, all function calls can be implemented as proper tail calls, making the stack not grow at all. Since all garbage collection roots are known (they consist of global variables and the arguments to the currently executed function), garbage collection can still be precise.

8.1 Acknowledgements

I thank Anton Ertl, Andreas Krall and Ulrich Neumerkel for their help and suggestions on this work.

I thank Philipp Tomsich for providing me with access to SPARC and PowerPC machines.

Appendix A

Tail-Annotated C Grammar

function-definition:
  declaration-specifiers declarator declaration-list$_{opt}$ *compound-statement*$_{tail}$

statement-wo-break$_{tail}$*:*
  labeled-statement$_{tail}$
  compound-statement$_{tail}$
  expression-statement$_{tail}$
  selection-statement$_{tail}$
  iteration-statement
  jump-statement

labeled-statement$_{tail}$*:*
  identifier : *statement*$_{tail}$
  `case` *constant-expression* : *statement*$_{tail}$
  `default` : *statement*$_{tail}$

expression-statement$_{tail}$*:*
  expression$_{tail\ opt}$;

selection-statement$_{tail}$*:*
  `if` (*expression*) *statement*$_{tail}$
  `if` (*expression*) *statement*$_{tail}$ `else` *statement*$_{tail}$
  `switch` (*expression*) *statement*$_{tail}$

compound-statement$_{tail}$*:*
  { *block-item-list*$_{tail\ opt}$ }

block-item-list$_{tail}$*:*
  block-item-wo-break$_{tail}$

`break ;`
`break ;` *block-item-list*$_{tail}$
block-item-wo-break$_{tail}$ `break ;`
block-item-wo-break$_{tail}$ `break ;` *block-item-list*$_{tail}$
block-item-wo-break block-item-list-wo-break$_{tail}$

block-item-list-wo-break$_{tail}$*:*
block-item-wo-break$_{tail}$ `break ;`
block-item-wo-break$_{tail}$ `break ;` *block-item-list*$_{tail}$
block-item-wo-break block-item-list-wo-break$_{tail}$

block-item-wo-break:
declaration
statement-wo-break

block-item-wo-break$_{tail}$*:*
declaration
statement-wo-break$_{tail}$

statement-wo-break:
labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement-wo-break

statement-wo-break$_{tail}$*:*
labeled-statement$_{tail}$
compound-statement$_{tail}$
expression-statement$_{tail}$
selection-statement$_{tail}$
iteration-statement
jump-statement-wo-break

jump-statement-wo-break:
`goto` *identifier* `;`
`continue ;`
`return` *expression*$_{tail\ opt}$`;`

jump-statement:
`goto` *identifier* `;`

`continue ;`
`break ;`
`return` *expression*$_{tail\ opt}$ `;`

expression$_{tail}$:
assignment-expression$_{tail}$
expression `,` *assignment-expression*$_{tail}$

assignment-expression$_{tail}$:
conditional-expression$_{tail}$
unary-expression assignment-operator assignment-expression

conditional-expression$_{tail}$:
logical-OR-expression$_{tail}$
logical-OR-expression `?` *expression*$_{tail}$ `:` *conditional-expression*$_{tail}$

logical-OR-expression$_{tail}$:
logical-AND-expression$_{tail}$
logical-OR-expression `||` *logical-AND-expression*

logical-AND-expression$_{tail}$:
inclusive-OR-expression$_{tail}$
logical-AND-expression `&&` *inclusive-OR-expression*

inclusive-OR-expression$_{tail}$:
exclusive-OR-expression$_{tail}$
inclusive-OR-expression `|` *exclusive-OR-expression*

exclusive-OR-expression$_{tail}$:
AND-expression$_{tail}$
exclusive-OR-expression `^` *AND-expression*

AND-expression$_{tail}$:
equality-expression$_{tail}$
AND-expression `&` *equality-expression*

equality-expression$_{tail}$:
relational-expression$_{tail}$
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

relational-expression$_{tail}$:
shift-expression$_{tail}$
relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*

shift-expression$_{tail}$:
additive-expression$_{tail}$
shift-expression `<<` *additive-expression*
shift-expression `>>` *additive-expression*

additive-expression$_{tail}$:
multiplicative-expression$_{tail}$
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

multiplicative-expression$_{tail}$:
cast-expression$_{tail}$
multiplicative-expression `*` *cast-expression*
multiplicative-expression `/` *cast-expression*
multiplicative-expression `%` *cast-expression*

cast-expression$_{tail}$:
unary-expression$_{tail}$
`(` *type-name* `)` *cast-expression*

unary-expression$_{tail}$:
postfix-expression$_{tail}$
`++` *unary-expression*
`--` *unary-expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof` `(` *type-name* `)`

postfix-expression$_{tail}$:
primary-expression$_{tail}$
postfix-expression `[` *expression* `]`
function-call$_{tail}$
postfix-expression `_` *identifier*
postfix-expression `->` *identifier*
postfix-expression `++`

postfix-expression `--`
`(` *type-name* `)` { *initializer-list* }
`(` *type-name* `)` { *initializer-list* `,` }

function-call$_{tail}$:
postfix-expression `(` *argument-expression-list*$_{opt}$ `)`

primary-expression$_{tail}$:
identifier
constant
string-literal
`(` *expression*$_{tail}$ `)`

Bibliography

[Ame99] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1999.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

[ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading, Mass.: Addison-Wesley, 1985.

[Bak95] Henry G. Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *ACM SIGPLAN Notices*, 30(9):17–??, September 1995.

[Bak97] Henry G. Baker. Garbage in/garbage out: When bad programs happen to good people. *ACM SIGPLAN Notices*, 32(3):27–31, March 1997.

[Bar89] Joel F. Bartlett. SCHEME->C A portable scheme-to-C compiler. Technical Report DEC-WRL-89-1, Digital Equipment Corporation, Western Research Lab, 89.

[BD95] Mark W. Bailey and Jack W. Davidson. A formal model and specification language for procedure calling conventions. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 298–310, New York, NY, January 1995. ACM.

[Bis93] Kurt M. Bischoff. User manual for ox: An attribute-grammar compiling system, based on yacc, lex, and C. Technical Report

TR92-30, Iowa State University, Department of Computer Science, December 1993.

[CD95] Philippe Codognet and Daniel Diaz. WAMCC: Compiling Prolog to C. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 317–332, Cambridge, June 13–18 1995. MIT Press.

[Com00a] Compaq Computer Corporation. Tru64 UNIX assembly language programmer's guide, 2000.

[Com00b] Compaq Computer Corporation. Tru64 UNIX calling standard for Alpha systems, 2000.

[Cri92] Régis Cridlig. An optimizing ML to C compiler. In *Proceedings of the 1992 ACM Workshop on ML and its Applications*, pages 28–36, San Francisco, U.S.A., June 1992. Association for Computing Machinery.

[DM94] B. Demoen and G. Maris. A comparison of some schemes for translating logic to c, 1994.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[FCRN94] T. Fujise, T. Chikayama, K. Rokusawa, and A. Nakase. Klic: A portable implementation of kl, 1994.

[Fee] Marc Feeley. Gambit-C. `http://www.iro.umontreal.ca/~gambit/`.

[Fou] The Free Software Foundation. GNU's not Unix! - the GNU Project and the Free Software Foundation (FSF). `http://www.gnu.org/`.

[GBD92] David Gudeman, Koenraad De Bosschere, and Saumya K. Debray. jc: An efficient and portable sequential implementation of Janus. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP-92)*, pages 399–416, Cambridge, November 9–13 1992. MIT Press.

[Hau94] Bogumil Hausman. Turbo Erlang: Approaching the speed of C. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1994.

[HCS95] Fergus Henderson, Thomas Conway, and Zoltan Somogyi. Compiling logic programs to C using GNU C as a portable assembler. In *ILPS'95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming*, pages 1–15, Portland, Or, 1995.

[Int00] Intel Corporation. Intel architecture software developer's manual—volume 1: Basic architecture, 2000.

[JNO98] S. P. Jones, T. Nordin, and D. Oliva. `C--`: A portable assembly language. *Lecture Notes in Computer Science*, 1467:1–??, 1998.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[PD82] S. Pemberton and M. C. Daniels. *Pascal implementation: the P4 compiler*. Ellis Horwood Limited, Chichester, England, 1982.

[SI95] Sun Microsystems, Inc. and IBM Corporation. System V application binary interface PowerPC processor supplement, 1995.

[Sil96] Silicon Graphics, Inc. MIPSpro N32 ABI handbook, 1996.

[Sis] Jeffrey Mark Siskind. Stalin. `http://www.neci.nj.nec.com/homepages/qobi/`.

[SPA92] SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[Sta99] Richard Stallman. *Using and porting the GNU Compiler Collection*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, 1999.

[Ste77] Guy Lewis Steele, Jr. Compiler optimization based on viewing LAMBDA as rename plus goto. Master's thesis, Massachusetts Institute of Technology, May 1977.

[Ste78] Guy Lewis Steele, Jr. RABBIT: A compiler for SCHEME (A study in compiler optimization). AI Technical Report 474, AI Lab, Massachusetts Institute of Technology, Cambridge, MA, May 1978. Revised version of [Ste77].

[SW95] Manuel Serrano and Pierre Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Proceedings of the 2nd Symposium on Static Analysis*, pages 366–381, Glasgow, September 1995.

[Tam94] Tanel Tammet. Hobbit: lambda-lifting as an optimization for compiling scheme. Draft, 1994.

[The] The Santa Cruz Operation, Inc. System V application binary interface SPARC processor supplement.

[TLA92] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

[Win] Felix L. Winkelmann. Chicken. `http://www.anu.ie/felix/chicken.html`.

[WR88] J. L. Weiner and S. Ramakrishnan. A piggy-back compiler for Prolog. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, pages 288–296, Atlanta, Georgia, 22–24 June 1988. *SIGPLAN Notices* 23(7), July 1988.