Compiling logic programs to C using GNU C as a portable assembler

Fergus Henderson, Thomas Conway, and Zoltan Somogyi
{fjh,conway,zs}@cs.mu.OZ.AU
Fax: +61 3 348 1184, Phone: +61 3 282 2401
Department of Computer Science, University of Melbourne
Parkville, 3052 Victoria, Australia

Abstract

This paper discusses the merits of using C, and in particular GNU C, as an intermediate target language for the compilation of logic programs, and describes the approach we have taken in the implementation of Mercury. We start with a simple approach using ANSI C, and investigate a variety of improvements on this basic approach.

Keywords: compilation techniques, programming language implementation, logic programming languages, Mercury, C, GNU C.

1 Introduction

There are many different ways of implementing a logic programming language, each with its own advantages and disadvantages. Each method makes its own trade-offs in terms of simplicity, portability, interactivity, compilation time, code size, speed at runtime, ease of debugging, interoperation with other languages, and so on. The choice of a particular method must be governed by the relative weights placed on these different trade-offs. This paper discusses the approach we took in the implementation of Mercury [11, 12], a new purely declarative logic programming language. We wanted an implementation that would be a useful and practical tool for building large applications. For our purposes in this paper, the two most important criteria were portability and efficiency of the generated code, and we chose to implement Mercury using C as an intermediate target language.

Although this paper is primarily about using C as an intermediate language, we use the Mercury implementation as an example, so section 2 gives some background information on Mercury. Section 3 explains why we chose to compile to C. Section 4 describes our basic compilation scheme using ANSI C, while section 5 discusses how we can take advantage of some GNU C extensions to improve efficiency. Section 6 discusses memory allocation. Section 7 gives a brief performance comparison.

For more information on the Mercury execution model, see [11, 12]; for more information on the Mercury code generator, see [4].

2 Mercury

Mercury is a new, purely declarative logic programming language. Like Prolog and other existing logic programming languages, it is a very high-level language that allows programmers to concentrate on the problem rather than the low-level details such as memory management. Unlike Prolog, which is oriented towards exploratory programming, Mercury is designed for the construction of large, reliable, efficient software systems by teams of programmers.

The main features of Mercury are:

- Mercury is purely declarative: predicates in Mercury do not have non-logical side effects.
- Mercury is a strongly typed language. Mercury's type system is based on many-sorted logic with parametric polymorphism, very similar to the type systems of modern functional languages such as ML and Haskell. Programmers must declare the types they need using declarations such as

```
:- type list(T) --->    [] ; [T | list(T)].
:- type maybe(T) --->   yes(T) ; no.
```

They must also declare the type signatures of the predicates they define, for example

```
:- pred append(list(T), list(T), list(T)).
```

The compiler infers the types of all variables in the program. Type errors are reported at compile time.

- Mercury is a strongly moded language. The programmer must declare the instantiation state of the arguments of predicates at the time of the call to the predicate and at the time of the success of the predicate. Currently only a subset of the intended mode system is implemented. This subset effectively requires arguments to be either fully input (ground at the time of call and at the time of success) or fully output (free at the time of call and ground at the time of success).

  A predicate may be usable in more than one mode. For example, append is usually used in at least these two modes:

  ```
  :- mode append(in, in, out).
  :- mode append(out, out, in).
  ```

  If a predicate has only one mode, the mode information can be given in the predicate declaration.

  ```
  :- pred factorial(int::in, int::out).
  ```

  The compiler will infer the mode of each call, unification and other builtin in the program. It will reorder the bodies of clauses as necessary to find a left to right execution order; if it cannot do so, it rejects the program. Like type-checking, this means that a large class of errors are detected at compile time.

- Mercury has a strong determinism system. For each mode of each predicate, the programmer should declare whether the predicate will succeed exactly once (det), at most once (semidet), at least once (multi) or an arbitrary number of times (nondet). These declarations are attached to mode declarations like this:

  ```
  :- mode append(in, in, out) is det.
  :- mode append(out, out, in) is multi.

  :- pred factorial(int::in, int::out) is det.
  ```

  The compiler will try to prove the programmer's determinism declaration using a simple, predictable set of rules that seems sufficient in practice (the problem in general is undecidable). If it cannot do so, it rejects the program.

  As with types and modes, determinism checking catches many program errors at compile time.

- Mercury has a simple module system.

- Mercury supports higher-order programming, with closures, currying, and lambda expressions.

For more information on Mercury, see the home page of the Mercury project at <http://www.cs.mu.oz.au/~zs/mercury.html>.

3 Why compile to C?

Since Mercury is in many respects similar to Prolog, one possible way — perhaps the simplest way — of implementing Mercury would be to write a translator from Mercury to Prolog. Indeed, we did start off by writing a very simple translator which translated a subset of Mercury to Prolog by just stripping away the additional declarations. However, although the semantics of Mercury are similar to those of pure Prolog, there are quite a few differences. For example, Prolog's if-then-else construct is non-logical, because it prunes away all but the first solution to the condition of the if-then-else, whereas Mercury's if-then-else does not prune away these solutions unless it is safe to do so.

It would be possible to write a more sophisticated translator which did the proper type, mode, and determinism checking required by Mercury and then translated to Prolog in a way that took the semantic differences between Mercury and Prolog into account. This sort of approach was taken in the implementation of Gödel [7], which despite significant syntactic differences is semantically a lot closer to Mercury than Prolog is. However, this approach would not allow us to achieve the performance we wanted. Right from the very start, high performance was a key goal for the Mercury implementation. Because Mercury has strong static type, mode, and determinism systems, a good Mercury compiler would be able to produce very efficient code by taking advantage of the type, mode, and determinism information that was available at compile time. An implementation that compiled to Prolog could not take advantage of this information.

Performance considerations also ruled out the use of an interpreter, or an implementation that compiled to byte-code and then interpreted the byte-code.

Compiling directly to assembler or machine code could potentially produce maximally efficient code, but would require an extremely large amount of work, and would make porting very difficult. A lot of this work would be reimplementing standard code generation and optimization techniques including instruction selection, instruction scheduling, and so on. The manpower we had available was not sufficient to do a good job of this for even a single architecture in the time available.

The natural choice which remained was compiling to C. Weiner and Ramakrishnan described an optimizing Prolog compiler which generated C code as early as 1988 [14]; like the Mercury compiler, it was designed to take advantage of information about types, modes, and determinism. However, unlike Mercury it appears that these annotations were not checked for correctness by the compiler, but were only for optimization. (An even earlier paper by Nilsson, from 1983, describes compilation into Pascal [10].)

With the widespread availability of good C compilers, and because C offers a variety of features that make it useful as a low-level target language, compilation to C has become an increasingly popular choice for language implementors. Programming language implementations that use C as an intermediate language include `wamcc` [3], `jc` (an implementation of Janus) [5], KL1 [2], Turbo Erlang [6], Gofer [8], `ghc` (the Glasgow Haskell compiler) [9], `cfront` (the original C++ implementation), several Eiffel implementations, and a variety of others. C is high-level enough to make it easy to generate code, but low-level enough to express low-level optimizations, and for C compilers to generate very efficient code.

4 Compilation to ANSI C

C does have some serious drawbacks as a target when compiling a logic programming language. The major problem is that C doesn't have any support for backtracking. Function calls in C obey a simple model in which local variables can be allocated on the stack on function entry, and deallocated on function exit. Backtracking in logic programming requires a more complicated model. When a non-deterministic predicate succeeds, control must return to the caller, but the storage for local variables must not be deallocated, since the predicate may be re-entered on backtracking.

This means that implementations of logic programming languages that work by compiling to C must do their own stack management, at least for non-deterministic predicates. Since C compilers typically don't do a very good job of last-call optimization, which is often very important for logic programs, it is also desirable for the implementation to do its own stack management even for deterministic predicates.

4.1 The Mercury abstract machine

The Mercury compiler compiles code first to a simplified parse tree, known as the HLDS (high level data structure), then to a low-level imperative intermediate form, known as the LLDS (low level data structure), and then transforms the LLDS into C code. The LLDS defines instructions for a simple abstract machine which we call the MAM (Mercury abstract machine).

The Mercury abstract machine has three main memory areas: the det stack, the nondet stack, and the heap. It has five special-purpose registers: `sp` (the det stack pointer), `maxfr` (which points to the topmost frame on the nondet stack), `curfr` (which points to the frame on the nondet stack used by the current procedure), `hp` (the heap pointer), and `succip` (the "success instruction pointer", which holds the return address). It also has a set of general-purpose registers `r1`, `r2`, `r3`, ..., which are used for argument passing and as temporaries.

The instruction set consists of assignments, labels, gotos, and conditional gotos, plus some other instructions that could in principle be synthesized from these. The source and destination of assignments can be complicated expressions involving constants, registers, arithmetic and comparison operations, pointer dereferencing, and the addresses of labels.

The Mercury abstract machine is discussed in more detail in our previous work [11, 12], which describes how the HLDS is transformed to the LLDS, and why the MAM has those particular memory areas and registers. This paper is concerned with how the LLDS code can be transformed to efficient C code.

4.2 The basic model

The Mercury implementation provides several different methods of implementing the Mercury abstract machine. All these methods map the various MAM registers onto a single set of general-purpose registers `mr0`, `mr1`, ... In the basic model, these "variables" are actually references to the elements of a single global array of "registers". (We use an array rather than independent global variables because the array can be pointed to by a single real register, which allows the C compiler to generate better code.) Mercury abstract machine expressions are mapped to C expressions in a very straightforward way. Implementing MAM labels and gotos is not so easy, however. They cannot be implemented as C labels and gotos, since in C it is not possible to take the address of a label. Instead, each labelled basic block in the Mercury abstract machine code is translated into a single C function. These C functions take no parameters, but return a continuation address. MAM gotos are translated into C return statements. A driver function in the runtime system, similar to the following simplified example, does the necessary dispatching to transfer control from one C function to the next:

```
typedef void * Func(void);
void driver(Func *entry_point) {
    register Func *fp = entry_point;
    while (fp != NULL) {
        fp = (Func *) (*fp)();
    }
}
```

In comparison with what could be achieved by compiling directly to machine code, the basic model is quite inefficient. Since the Mercury abstract machine registers are represented as global variables, accessing them is a lot more expensive than accessing real machine registers would be. The cost of MAM gotos is one C function call and return plus one iteration of the driver loop — in other words, a test and three or four branches, plus some stack manipulation and so forth in the C function prologue and epilogue.

The loop termination test `fp != NULL` can be eliminated by using `exit()` or `longjmp()` to exit from the loop. A little care is required to make the driver function reentrant, so as to allow reentrant interfacing to C. In the Mercury runtime, we use `longjmp(*jmp_buf_ptr)` to exit the driver loop, where `jmp_buf_ptr` is a global variable which points to a `jmp_buf` which is a local variable in the driver function. To ensure reentrancy, the driver function saves the value of `jmp_buf_ptr` on entry and restores the old value on exit. This means we can allow C code called from Mercury code to call other Mercury code in turn.

```
jmp_buf *jmp_buf_ptr;
typedef void * Func(void);
void driver(Func *entry_point) {
    register Func *fp = entry_point;
    jmp_buf* save_jmp_buf_ptr = jmp_buf;
    jmp_buf *save_jmp_buf_ptr = jmp_buf_ptr;
    jmp_buf local_jmp_buf;

    jmp_buf_ptr = &local_jmp_buf;
    if (setjmp(local_jmp_buf) == 0) {
        while (1) {
            fp = (Func *) (*fp)();
        }
    }
    /* we arrive here only after a ‘longjmp(*jmp_buf_ptr)’ */
    jmp_buf_ptr = save_jump_buf_ptr;
}
```

The cost of the dispatch loop can be reduced a little further by unrolling:

```
    while (1) {
        fp = (Func *) (*fp)();
        fp = (Func *) (*fp)();
        fp = (Func *) (*fp)();
        fp = (Func *) (*fp)();
        fp = (Func *) (*fp)();
        fp = (Func *) (*fp)();
        fp = (Func *) (*fp)();
        fp = (Func *) (*fp)();
    }
```

This reduces the overhead of the loop from one branch per iteration to one branch per eight iterations, and thus cuts the overall cost to a little more than two or three jumps per MAM goto, plus the function prologue and epilogue code.

One significant drawback of the basic model model is that it is very hard for the C compiler to optimize the code well, since the use of the driver loop makes it effectively impossible for the C compiler to analyze the control flow, and the use of global variables makes it very difficult for the C compiler to analyze the data flow. Furthermore, the large number of function prologues and epilogues also has a significant detrimental impact on code size.

Another improvement is possible if a MAM label is only referred to from within its own block and its address is not taken. One can then translate this MAM label into an ordinary C label, and an MAM goto can be implemented as a C goto. This technique can also be applied to several consecutive labels if they have the property that all gotos to those labels are from within the blocks started by those labels. However, it is necessary to take the address of the label following each procedure call (other than tail calls) so that when the called procedure has completed it can jump to that return address. Hence the potential improvement from this optimization is limited.

Version 0.5 of the Mercury compiler performs this optimization for the single-label case, but the implementation is different. The label is translated into `while (1) {`, and gotos to the label are translated into `continue`. To prevent undesirable looping, `break; }` is inserted at the end of the block before the next label.

What we have called the basic model is basically the same as the techniques used in Gofer [8].

5 Compilation to GNU C

One way of improving efficiency is to take advantage of the features of GNU C. The GNU C compiler [13], `gcc`, is free software. It generates reasonably efficient code, it is very widely available, and it has been ported to many platforms. GNU C provides a variety of extensions over standard ANSI C, some of which are very useful if you are using it as a target language, since they allow you to generate more efficient code.

The extensions offered by gcc include

- the ability to make direct use of the machine registers using global register variable declarations;
- the ability to take the address of labels, and to later jump to those addresses; and
- the ability to insert inline assembler code, and to specify the assembler name for a function.

In the sections below, we explain how each of these extensions can be used to improve efficiency.

These extensions are a standard part of gcc and are available on all gcc ports. Nevertheless, despite gcc's widespread availability, relying on these extensions would reduce the portability of the code we generate. Rather than do this, we decided to use conditional compilation (i.e. `#ifdef`) so that these features are only exploited if particular macros (`USE_GCC_GLOBAL_REGISTERS`, `USE_GCC_NONLOCAL_GOTOS`, and `USE_ASM_LABELS`) are defined. As far as we know, our compiler is the first to emit C code that, with appropriate definitions of macros, can be compiled to either to exploit or not to exploit each particular GNU C extension.

For example, local labels are declared and defined with macros `Declare_label(label)` and `Define_label(label)`. To take the address of a label, we use `LABEL(label)`, and to jump to such an address, we used `GOTO(address)`. In the basic model, these macros are defined as follows.

```
#define LABEL(label)            (label)
#define GOTO(address)           return (address)
#define Declare_label(label)    static Code *label(void)
#define Define_label(label)     \
                GOTO(label);    \
        }                       \
        static Code* label(void) {
```

The `Define_label` macro ends the current function and starts a new one. It includes a `GOTO` macro to ensure that if control flow drops through to the label, it will continue in the new function.

We use some similar macros to define labels that will be used as local or exported procedure entry points; in the basic model, the only difference is that for exported predicates, these use `extern` rather than `static`. The code to begin and end a section of generated code is also macroized.

5.1 Global register variables

GNU C allows the programmer to declare that certain global variables occupy specific machine registers. For example, the declaration

```
register int x __asm__("ebx");
```

specifies that the variable x is really just a name for the machine register named "ebx". Since the registers which can be used vary between different architecture, code that uses global register variables must be conditional on CPU type, but this is easy to accomplish.

If USE_GCC_GLOBAL_REGISTERS is defined, and GNU C global registers variables are available on the particular platform in question, then we arrange for the first few virtual registers (mr0, mr1, ...) to be declared as global register variables. For example, here's the code we use for the i386 architecture.

```
#define NUM_REAL_REGS 3
register Word mr0 __asm__("esi");
register Word mr1 __asm__("ebx");
register Word mr2 __asm__("edi");
```

The number of global register variables we can use depends on the hardware architecture and the configuration of gcc. To simplify calls to C library functions, at the moment we exploit only registers that gcc designates to be callee-save; this means that we do not have to save and restore the registers before each call to a C function. We use 3 registers on x86s, 7 ($9–$14) on Alphas, 8 (s0–s7) on MIPS processors, and 10 (i0–i5 and l1–l4) on SPARCs.

The use of global register variables on SPARCs is complicated by the SPARC's sliding register windows. The SPARCs registers are divided into 4 groups of 8: global registers, input registers, local registers, and output registers. Only the global registers retain the same value in the caller and callee of a C function; the other groups are remapped, so that on function entry, the output registers become the new input registers, and on function exit, the input registers and local registers are restored. However, of the SPARC's 8 global registers, one is hardwired to zero, g1-g4 are caller-save, and g5-g7 are "reserved for the operating system" and get clobbered by the assembler routines which do integer multiplication, division, and modulus. Unlike wamcc [3], which uses g1, g5, g6, and g7, and carefully avoids the use of integer multiplication, division, and modulus (it does those operations using floating-point), we use the local and input registers. We don't have to worry about calls to C functions such as printf(), since the allocation of a new window of registers for the called function will protect the registers used by our virtual machine. We do have to make sure that for the few MAM operations which are are implemented as function calls, not macros, and which need access to the virtual machine registers, the function call is surrounded by code that copies the registers to global variables and then back to registers where necessary. We use a pair of macros save_transient_registers() and restore_transient_registers() to do this. For machines without register windows, these macros are defined to do nothing.

The most frequently-used MAM registers are mapped to lowest numbered virtual registers. We obtained accurate register-use statistics by instrumenting the code generated for real Mercury programs with instructions to count the number of uses of each register. This was done with more use of conditional compilation; the Mercury implementation contains code similar to the following:

```
#ifdef __GNUC__
  #define LVALUE_SEQ(expr, lval)        ((expr),(lval))
#else
  #define LVALUE_SEQ(expr, lval)        (*((expr),&(lval)))
#endif

#ifdef MEASURE_REGISTER_USAGE
  #define count_usage(num,reg)          LVALUE_SEQ(num_uses[num]++, reg)
#else
  #define count_usage(num,reg)          (reg)
#endif

#define sp          count_usage(0, mr0)
#define succip      count_usage(1, mr1)
```

No.	Reg	Usage	Cumulative usage
1	sp	32.63%	32.63%
2	succip	12.09%	44.71%
3	r1	11.87%	56.58%
4	r2	11.20%	67.78%
5	r3	10.49%	78.27%
6	r4	8.67%	86.94%
7	r6	4.49%	91.43%
8	r5	4.03%	95.46%
9	r7	1.97%	97.43%
10	r8	0.94%	98.36%
11	r9	0.47%	98.84%
12	r10	0.27%	99.11%
13	r11	0.19%	99.30%
14	hp	0.19%	99.49%
15	curfr	0.17%	99.66%
16	r12	0.16%	99.82%
17	r13	0.07%	99.89%
18	maxfr	0.06%	99.95%
19	r14	0.02%	99.97%
20	r15	0.02%	99.98%
21	r16	0.00%	99.99%
22	r18	0.00%	99.99%
23	r17	0.00%	100.00%
24	r19	0.00%	100.00%
25	r20	0.00%	100.00%
26	r21	0.00%	100.00%

Table 1: Register usage counts for the Mercury compiler

```
#define r1          count_usage(2, mr2)
#define r2          count_usage(3, mr3)
...
```

The `LVALUE_SEQ` macro is needed so that registers can be used as lvalues (e.g. on the left hand side of an assignment). `LVALUE_SEQ(expr,lval)` has the effect of executing `expr` (for its side effects), and then returning `lval` as an lvalue. In ANSI C, a comma expression is not an lvalue, so the `*((expr),&(lval))` version must be used. But it is not possible to take the address of a register, so this won't work with GNU C if `lval` is a global register variable. Fortunately in GNU C comma expressions are lvalues, so the simple `(expr),(lval)` version can be used.

Table 1 shows the register-use statistics for a sample execution of the Mercury compiler itself. On the SPARC, with 10 real registers available, 98% of accesses to virtual registers are accesses to real registers. This is similar to the figures reported for `jc` [5], where 10 real registers accounted for 95% of the register references.

On the 386, with only 3 real registers available, the figure is only 57%. For the 386, we could probably do a little better if we used callee-safe registers, or if we were compiling directly to assembler or machine code, but the 386 only has 8 "general-purpose" registers, and some of these are needed for other purposes such as temporary expression evaluation anyway, so we probably couldn't do that much better.

5.2 Taking the addresses of labels

Using GNU C's global register variables provides a quite reasonable solution to the efficiency problems caused by using C global variables for the abstract machine registers. However, we also need to address the efficiency costs associated with implementing each basic block as a separate function with a driver loop to transfer control between different basic blocks.

As noted in Section 4.2, by carefully optimizing the driver loop, the overhead of a MAM goto can be reduced to just over that of one C function return and call. Nevertheless, this is still a lot higher than the cost of a single jump instruction. Furthermore, as noted earlier the C function prologues and epilogues consume quite a bit of code space, and the consequent reduced locality leads to more instruction cache misses.

Fortunately, GNU C has another extension which can be used to address this problem. In GNU C, you can take the address of a label using a prefix `&&` operator, and then later jump to it using a "computed goto" statement such as `goto *return_address`. This means that MAM gotos and labels can be implemented as GNU C gotos and labels, avoiding the problems caused by the driver loop technique. So when `USE_GCC_NONLOCAL_GOTOS` is defined, the macros mentioned at the start of Section 5 are defined as follows.

```
#define LABEL(label)          (&&label)
#define GOTO(address)         goto *(address);
#define Declare_label(label)  /* no declaration required */
#define Define_label(label)   label:
```

One problem with generating code using these macros is that GNU C makes overly conservative assumptions about labels whose addresses is taken, and as a result generates worse code for `goto *(&&label)` than for `goto label`. To avoid this, the Mercury compiler is careful to generates calls to a specialized macro `GOTO_LABEL(label)` rather than `GOTO(LABEL(label))`. In the basic model `GOTO_LABEL(label)` is just the same as `GOTO(LABEL(label))`, but when using non-local gotos, it is defined as `goto label`, which keeps gcc's optimizer happy.

Unfortunately there are some technical difficulties with the general technique of using non-local gotos. According to the GNU C manual, "totally unpredictable things will happen" if computed gotos jump from one function to code in a different function.

The obvious way to avoid this would be to put all the code in a single function, and that is the approach taken by `jc` [5]. But that approach prevents separate compilation, and in any case it is impractical for any but the smallest of programs, since many of gcc's optimizations take time and space that is quadratic (or worse) in the size of the function being optimized. (Disabling optimization would not help, since the loss of optimization would lead to less efficient code than simply using the driver loop technique.) Since we wanted Mercury to support programs of hundreds of thousands of lines of code, we did not consider that option to be feasible.

However, considerable inspection of the gcc source code shows that despite what the GNU C manual says, in most circumstances it is possible to use inter-function jumps quite safely provided you take suitable precautions. This technique is not 100% portable, but we have used it quite successfully in the Mercury implementation on a variety of systems, and the gains in efficiency make it well worth the effort, especially since our use of conditional compilation means that we always have the driver loop technique to fall back on for those few systems for which this technique doesn't work.

The basic problem with inter-function jumps is that on entry to the middle of a function via an inter-function jump, the stack frame and the registers may not have been correctly set up in the way that the C compiler expects. The main precautions necessary to avoid trouble are (a) making sure that none of the functions involved has any local variables and (b) making sure that the code has access to a big enough stack frame to hold any spilled temporary values, by allocating a large stack frame in the driver function.

This technique does run into trouble with position-independent code on CPUs which do not support PC-relative addressing. On such CPUs the usual way C compilers generate position-independent code is to reserve one register, often called the `gp` (global pointer) register, to hold a context pointer. All accesses to global variables are done via this register. Before calling a function, the register is set to the address of the function; the function prologue code can then adjust the value of the `gp` register based on the offset of the function from its global data (this offset is known at static link time) so that the gp register points to the start of the global data. The gp register is also saved onto the stack and then restored after each function call. Since inter-function jumps don't set the gp register to the correct value, it is not possible to combine the use of position-independent code and inter-function jumps on such machines.

On DEC Alphas running OSF/1, position-independent code is mandatory – all code must be PIC, regardless of whether it is in a shared library or whether it uses shared libraries — and so inter-function jumps cannot be used. On SGI machines running IRIX 5, position-independent code is not mandatory, but is required for code that uses shared libraries, and for code that is to be placed inside a shared library. On IRIX 5 machines Mercury allows both options; shared libraries are used by default, but inter-function jumps can be used for programs such as the Mercury compiler itself for which the performance gain is more important than the use of shared libraries. On SPARCs running Solaris, position independent code is not required, although shared libraries that don't use PIC must be linked at runtime, and this prevents runtime sharing of the code pages of such libraries. Nevertheless, for Mercury the performance benefit of inter-function jumps more than makes up for this, and the slightly slower startup can always be avoided by linking statically if necessary. On DECstations running ULTRIX and on x86 PCs running BSDI BSD/386 1.1, the OS does not support shared libraries and position-independent code is not used, so the problem does not arise. On x86 PCs running Linux, the situation would be similar to that with IRIX 5, except that the conservative garbage collector that is used by the Mercury compiler does not yet support shared libraries on Linux, and so Mercury just uses inter-function jumps.

In summary, on four of the six platforms that we have ported Mercury to, the technique of using inter-function jumps does not cause any trouble; on one it interferes with the use of shared libraries; and on one it is not possible.

It might be possible to use inline assembler or some other trick to ensure that the gp register is correctly set up before doing inter-function jumps, but doing this does not appear to be easy, if it is possible at all.

When exploiting GNU C's nonlocal gotos, one must decide how much code should be inside each C function. The Mercury compiler has an option whereby the programmer can request that everything in a Mercury module should be put inside one C function. This is good for speed, since jumps within a C function may be faster than jumps between C function, and is good for code size, since it minimizes the number of function prologues and epilogues. However, as noted by other researchers, using very large functions can cause gcc to slow down significantly and to require much more memory. So by default, the Mercury compiler emits several C functions for each Mercury module, with each C function containing at most a set number of Mercury procedures. (This number is also configurable using a command-line option.)

5.3 Assembler labels

Using GNU C nonlocal gotos to jump between different C functions requires code in each function to know the addresses of labels in other functions. This can be done in one of two ways. First, one can arrange for each C function to start with code that takes the addresses of its own labels, puts them into global variables, and then returns. Cross-function jumps can get the addresses they need from these variables after an initialization routine has called all these functions at startup. The major disadvantage of this approach is that it requires paging in almost every page of the executable at program startup, to execute these initializations; this can lead to very significant startup delays for large applications.

To avoid this problem, we generate assembler labels directly into the code using inline `asm` code. The inline assembler consists only of `.globl` directives and labels, which seem to be quite portable — so far, they have worked on all the platforms we have ported to, without requiring any platform-specific `#ifdefs`. (The only `#ifdef` in this part of the code is the `#ifdef USE_ASM_LABELS`, which we use so that we can turn disable the use of assembler labels if necessary.)

We also use another GNU C extension to associate a C function name with the assembler name,

```
extern void func(void) __asm__("asm_name");
```

so that we can jump to the assembler label using just `goto *func`, without having to depend on the vagaries of how C function names are mapped to assembler label names (i.e. on whether or not C function names get underscores prepended to them) for the particular platform.

So in summary, if `USE_ASM_LABELS` is defined, the macros for local labels remain unchanged, but the corresponding macros for exported procedure entry points are defined as follows.

```
#define stringify(string) #string

#define Declare_entry(label)        \
        extern void label(void) __asm__("entry_" stringify(label))
#define Define_entry(label) \
        }       \
        label:  \
                __asm__(".globl entry_" stringify(label) "\n"   \
                        "entry_" stringify(label) ":"); \
        {
#define ENTRY(label)        (&label)
```

These macros provide the same role for external entry points as the `Declare_label`, `Define_label`, and `LABEL` macros do for local labels.

If the only reference to a piece of code is via an assembler label, the compiler may not realize that the code is reachable, and may optimize it away. To avoid this, we make sure all reachable labels are referenced. Originally, we did this by arranging for the code at the start of each function to assign the addresses of such labels to a volatile global variable. But now we instead use a dummy asm statement that makes use of GNU C's extended inline asm.

```
        __asm__("" :: "g"(&&label));
```

The empty string (`""`) means the dummy asm statement has no assembler code, but the `::"g"(&&label)` part means that it supposedly takes `&&label` as an input. We use this subterfuge to tell gcc that the label is used, thus preventing an over-zealous gcc from optimizing away `label` and the code that followed. This technique is better than assigning to a global variable, because gcc doesn't generate any code for the dummy asm statement.

6 Memory allocation

The traditional way to allocate memory for the storage of terms is for the runtime system to define a large array and an associated variable, the heap pointer, which initially points to the start of the array. Allocating memory consists of incrementing the heap pointer and checking for overflow; the pointer to the newly allocated block is then usually tagged.

On modern versions of Unix, the overhead of checking for heap overflow can be eliminated through the use of the mprotect system call, which allows the language implementation to ask the operating system to make the page at the end of the array unaccessible. When the program dereferences a pointer into this so-called "red zone", the kernel will send a signal to the process to notify it of the violation; the runtime system can then either terminate the program or initiate garbage collection.

The easiest way to implement a garbage collector is to reuse someone else's. Hans Boehm and his group have made available their conservative garbage collector for C [1]. Since this package does not know which words contain pointers and which contain other kinds of values, it cannot adjust pointers and therefore performs no compaction. Like other conservative collectors, this system considers any word in

system	variant	mean
SWI		1.00
NU	no declarations	1.72
	declarations	1.67
wamcc		3.14
Quintus		3.74
SICStus	compactcode, no declarations	2.02
	compactcode, declarations	1.88
	fastcode, no declarations	6.83
	fastcode, declarations	3.15
Aquarius	no declarations, no global analysis	9.87
	declarations, no global analysis	12.26
	no declarations, global analysis	18.42
	declarations, global analysis	19.00
Mercury	none grade, Boehm gc	6.36
	asm_fast grade, Boehm gc	9.24
	none grade, no gc	13.39
	asm_fast grade, no gc	36.52

Table 2: Averages of benchmark speed ratios

the address space containing the address of a cell to be a reference to that cell, even if it represents an integer or floating point value that happens to have the same bit pattern. The package can be told that if there is a cell at (say) the address 0x400, it should consider occurrences of the bit patterns 0x401, 0x402 and 0x403 to be references to the cell as well. This is very useful for implementations that store tags in the low order bits of pointers.

The Boehm garbage collector uses its own implementation of malloc. Although this `gc_malloc` has been carefully tuned, it is still a general-purpose memory allocator, and it is significantly slower than a simple increment of a heap pointer. To make the common case of allocating many small objects fast, the Boehm collector keeps an array of free lists of different sizes. The system also allows calls to `gc_malloc` to be inlined, so the common case reduces to just checking if the free list is empty, removing the first entry from the free list, and incrementing a counter of the number of words allocated. Nevertheless, for very highly allocation-intensive benchmarks such as nrev, this is still a lot slower than a single register increment. For real programs, garbage collection is not likely to be a bottleneck, and inlining `gc_malloc` would most likely reduce performance, as the elimination of the call overhead would be outweighed by the increase in instruction cache misses.

The other drawback of the Boehm collector, which applies only to logic programming systems, is that one cannot save the heap pointer when entering a choice point and restore it upon backtracking. Since blocks that are allocated consecutively in time are not necessarily consecutive in space, the overhead of keeping track of the blocks allocated since a certain point in time would be more than the possible gain in all except the most heavily nondeterministic programs.

The Mercury implementation currently has two implementations of the macro that allocates memory on the heap: one increments a heap pointer without an overflow check, relying on a red zone to catch overflow, and one that calls `gc_malloc`. We are working on our own native garbage collector, which will be invoked by the red zone signal handler.

7 Performance

We measured the performance of several logic programming implementations on a suite of ten standard small benchmarks; converted the raw times to speedups over SWI-Prolog, and calculated the harmonic mean of these speedups. The results are shown in table 2 (reproduced from [12]). The benchmark platform was a SPARCserver 1000 with four 50 MHz SuperSPARC processors, each rated at 60.3

System	variant	run time
NU-Prolog	no declarations	116 s
SICStus	compactcode, no declarations	101 s
SICStus	fastcode, no declarations	71.1s
Mercury	asm_fast grade, Boehm gc	18.5s

Table 3: The Mercury compiler as a large benchmark

SPECint92.

The results show that the computational model of the language and the amount of effort put into optimization are more important for good performance than the compilation target (bytecode, C or machine code). Mercury being the fastest system does not prove the superiority of compiling to C, since wamcc also compiles to C, and also exploits all three GNU C extensions we discussed, yet it is slower than Quintus Prolog, a bytecoded system. On the other hand, our data does not prove the superiority of compiling directly to native code either.

Our data does show the usefulness of the GNU C extensions. The "none" grade of the Mercury compiler uses none of the extensions, whereas the "asm_fast" grade uses all three (global register variables, inter-function jumps, and assembler labels); the asm_fast grade is 170% faster. Assembler labels are irrelevant to small benchmarks, since their code fits in a single C function; this extension benefits only large programs. Of the two remaining extensions, our experience suggests that global register variables improve performance significantly more than inter-function jumps do, but one cannot measure this on SPARCs because one cannot use any of the registers affected by register window operations as global register variables.

The results also show that although the use of Boehm's conservative garbage collector is convenient, it does impose significant overheads. These overheads are exaggerated by our small benchmarks. To obtain accurate times, each benchmark had to be executed thousands, even millions of times, in a test harness that effectively mimics a failure-driven loop. Since the Boehm collector cannot recover memory on backtracking, it does a significant number of otherwise useless collections. This effect is much less significant in real programs, as shown by our experiments using the Mercury compiler itself as a benchmark program.

The Mercury compiler is written in the intersection of three languages, Mercury, NU-Prolog and SICStus Prolog, and can be executed by any one of these systems. (We implemented the compiler this way to ease bootstrapping.) Table 3 shows the time taken by the Mercury compiler to compile a 400 line source file, when the compiler is executed by each of these three systems. While the small benchmarks show the asm_fast grade of Mercury with Boehm gc to be 1.35 times the speed of SICStus fastcode, the large benchmark shows the ratio to be 3.84. Comparing Mercury asm_fast with Boehm gc to with SICStus compactcode yields broadly similar although not so striking results (a speedup of 4.57 on the small benchmarks vs 5.46 on the large program) as does a comparison against NU-Prolog (5.37 vs 6.27). This shows that at least some large programs (e.g. the Mercury compiler) are not as allocation intensive as the standard small benchmarks we used (crypt, deriv, nrev, poly, primes, qsort, two versions of queens, query, and tak), and therefore the use of a non-compacting conservative garbage collector is not as bad it may seem at first sight.

NU-Prolog produces a 2.1 Mb save file when compiling the Mercury compiler. SICStus compactcode produces a 4.8 Mb save file, while SICStus fastcode produces an 8.1 Mb executable. Mercury produces a 2.2 Mb executable, which shows that compilation to C per se need not incur any space penalties. Of course, the main reason for the small size of the Mercury executable is that the Mercury execution model simply does not need many of the operations that are required to execute Prolog (dereferencing, trailing etc), and can therefore omit the code to perform these operations.

8 Conclusions and further work

Our performance results are too coarse to argue conclusively either the superiority of native code compilation over compilation to C or vice versa when the system that compiles to C exploits GNU C extensions. Our opinion is that compilation to GNU C loses a few percent of speed compared to a highly optimizing compiler that generates machine code directly, but that this consideration is much less important than the advantages enjoyed brought by compilation to C. These include the fact that much less work is required on the part of compiler writers, who can spend the extra time on other aspects of the system, and the greatly increased portability of the resulting system. We have shown that portability can be maximized through a modular approach to the implementation of the virtual machine that makes intensive use of C macros.

Our paper shows ways to compile logic programs to C using C as a portable assembler. One can also generate C code in a manner that uses C as a higher level language. Weiner and Ramakrishnan [14] have described an alternative approach using continuations, with assembler routines to construct a closure and to execute a closure. We are currently investigating a similar continuation-based approach using GNU C's nested functions extension for constructing and executing the closures, as an alternative to the approach that we currently use, which we have described in this paper. This new approach, which has some similarities to the approach taken by Nilsson in a paper that proposed the compilation of Prolog to Pascal [10], has the potential to make debugging with gdb a practical proposition.

References

[1] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.

[2] T. Chikayama, T. Fujise, and D. Sekita. A portable and efficient implementation of KL1. In *ICOT/NSF Workshop on Parallel Logic Programming and its Programming Environments*, CIS-TR-94-04, Department of Computer Information Science, Oregon, 1994.

[3] P. Codognet and D. Diaz. wamcc: Compiling Prolog to C. In *Proceedings of the Twelfth International Conference on Logic Programming*, pages 317–331, Kanagawa, Japan, June 1995.

[4] T. Conway, F. Henderson, and Z. Somogyi. Code generation for mercury. In *Proceedings of the Twelfth International Conference on Logic Programming*, Portland, Oregon, December 1995.

[5] D. Gudeman, S. K. Debray, and K. DeBosschere. jc: an efficient and portable sequential implementation of Janus. In *Proceedings of the International Conference and Symposium on Logic Programming*, pages 399–416, Washington, D.C., November 1992.

[6] B. Hausman. Carpe diem, some implementation aspects of Turbo Erlang. In *Proceedings of the Workshop on Practical Implementations and Systems Experience in Logic Programming*, pages 1–12, Budapest, Hungary, June 1993.

[7] P. M. Hill and J. W. Lloyd. *The Gödel programming language.* MIT Press, 1994.

[8] M. P. Jones. The implementation of the Gofer functional programming system. Technical Report Research Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA, May 1994.

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

[10] J. F. Nilsson. On the compilation of a domain-based Prolog. In *Proceedings of the Ninth IFIP Congress*, pages 293–298, Paris, France, 1983.

[11] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming.* To appear.

[12] Z. Somogyi, F. Henderson, and T. Conway. The implementation of Mercury, an efficient purely declarative logic programming language. In *Proceedings of the ILPS '94 Postconference Workshop on Implementation Techniques for Logic Programming Languages*, Syracuse, New York, November 1994.

[13] R. Stallman. *Using and porting the GNU CC compiler.* The Free Software Foundation, 1989–1995.

[14] J. Weiner and S. Ramakrishnan. A piggy-back compiler for Prolog. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 288–296, Atlanta, Georgia, June 1988.