A report on the progress of GNU Modula-2 and its potential integration into GCC

Gaius Mulley
University of Glamorgan
gaius@glam.ac.uk

Abstract

This paper reports on the status of the GNU Modula-2 front end to GCC and the extensions made to Modula-2 and gdb to ease its potential integration into the main GCC source tree. GNU Modula-2 (`gm2`) is maturing into a reliable tool and it now builds and passes its regression tests on the following platforms: x86, Opteron, Athlon 64, Alpha, Itanium processors running GNU/Linux, Sparc based Solaris, PowerPC MacOS, x86 Open Darwin and the x86 processor running FreeBSD. GNU Modula-2 currently conforms to all three Programming in Modula-2 dialects as defined by Wirth.

The paper also describes the two categories of language extensions made. The first category follows the tradition of other GCC front ends by allowing the in-lining of assembly language, conditional compilation, procedure inlining and allowing users to cleanly exploit the GCC library of built-in functions and constants. The second category provide easy access to C libraries. The work presented here discusses the portable implementation of open arrays, module priorities, coroutine primitives and multi-word sets. It also reports on many of the key design decisions taken during the construction of GNU Modula-2 and their various implications.

1 Introduction

Source code which cannot port to another architecture will die [Gancarz]. The motivation for producing a Modula-2 front end for GCC includes providing a robust compiler for production systems, providing a migration path for legacy source code and producing a compiler which can generate enhanced semantic error messages for student programmers.

Modula-2 is a relatively small language in contrast to C++ and Ada yet its flexibility with low level processes, bit manipulation, interrupt handling make it an ideal language with which to implement small footprint embedded systems. Its other strengths are an enforced modularity, the abstract data type and its distinctive definition and implementation module files. It also found favour amongst many academic undergraduate teaching programs during the 1980s and 1990s. Some eminent academics argue that a language, of a similar simplicity to Modula-2, is ideal for teaching new students to program. [1]

Currently there are only a few commercial Modula-2 compilers being actively maintained. Code which was written ten or fifteen years ago may still be compiled by older commercial (possibly unmaintained) Modula-2 compilers,

[1] `http://www.iticse2002.dk/conference/Talk/iticse2002.html`

however a number of these compilers generate 16 bit code. While the 32 bit x86 processors remain, compilers targeting these processors may be run in compatibility mode. Time is running out as the computing industry is switching to 64 bit microprocessors [AMD64] [Opteron] [Intel1] [IBM970]. While x86 emulation, 16 bit backwards compatibility or running 32 bit code on a 64 bit platform are all possible they all have serious drawbacks. In order for the older source to be compiled into a native executable it will either have to be translated into another high level language or alternatively a Modula-2 compiler which can target these new generation of microprocessors will have to be acquired. GNU Modula-2 suits this purpose as it has the advantage of being closely tied to GCC. Not only does this produce excellent code and good architectural and operating system coverage but it also utilises many of the GCC features. For example GNU Modula-2 can: invoke the C preprocessor to manage conditional compilation; in-line `SYSTEM` procedures, intrinsic functions and memory copying routines; provide access to assembly language using the GCC syntax.

GNU Modula-2 currently supports all dialects of *Programming in Modula-2* as defined by Wirth [Wirth1] [Wirth2] [Wirth3] and work is underway to support the ISO dialect [ISO]. It enhances a number of language features, for example: it allows sets to be declared of any ordinal type; abstract data types are not restricted to a pointer type in the implementation module; constants, types, variables may be declared in any order. The compiler provides numerous command line options which: enable runtime and compile checking, specific optimisations, runtime behaviour of `DIV` and `MOD` operators as well as library dialect and various linking options.

Given that the original C compiler has become the GNU compiler collection it is fitting that a Modula-2 front end should exist.

2 Previous work

There was a previous GNU Modula-2 effort undertaken by the computer science department at the State University of New York at Buffalo [Bowen]. A substantial amount of work was achieved but funding terminated before the compiler was complete. In particular Modula-2 support was added to the GNU debugger `gdb`. The compiler had an elegant method of interfacing to C through the use of the keywords `DEFINITION MODULE FOR "C"` in the definition module. This front end matched GCC release 2.3.3 in 1994. Since then the GCC internals have changed substantially to incorporate a different paradigm of garbage collection.

Previously at the University of Glamorgan a Modula-2 compiler (`m2f`) was produced which performed detailed semantic checks and informative error messages [Mulley] [Lewis]. A number of these checks were performed post intermediate code optimisation in an attempt to maximise the knowledge the compiler had about the program source. This resulted in the compiler having the ability to detect elementary infinite loops.

3 Goals of GNU Modula-2

GNU Modula-2 will support the language as defined by Wirth [Wirth1] [Wirth2] [Wirth3] in “Programming in Modula-2” (PIM) and also the ISO Modula-2 standard [ISO]. The GNU Modula-2 project has opted for a release early strategy [Raymond] and it will initially support the PIM dialect of the language before implementing the ISO standard. The differences between the last three PIM editions are so small that they can all be incorporated into one compiler and they may be individually selected by command line options.

The GNU Modula-2 implementation must respect the fact that Modula-2 allows programmers to declare types, variables, constants and procedures in any order. Also the compiler must not include any artificial programming limits [Pronk] and therefore the implementation must avoid fixed array sizes and use dynamic data structures throughout [Stallman2].

Given that modern software projects are unlikely to be completely written in Modula-2 and a large target audience of `gm2` will be maintainers of legacy software it is vital to include good access to other languages. This is crucial since it is a core aim that `gm2` will be one component of the GNU compiler collection. A clean interface between Modula-2 and C also aids the development of `gm2`. There will be two types of extensions to Modula-2 the first facilitates access to other languages and the second provides features found in the other GNU compilers.

4 Extensions to Modula-2

There is no mechanism to manage conditional compilation in any of the four language specifications of Modula-2 mentioned earlier. The GNU implementation of Modula-2 allows the C preprocessor to be invoked when the option `-Wcpp` is present on the command line. This option tells the C preprocessor to operate in traditional mode, using assembler as a base language and it preserves comments and pays no attention to single quote or double quote characters. Thus macro argument symbols are replaced by the argument values even when they appear within string or character constants. It also ensures that the symbols `#` and `##` have no special meaning. This strategy also matches the behaviour of another GNU compiler front end (namely `f77`).

GNU Modula-2 allows local procedures to be compiled in-line. Currently the programmer has to declare a procedure using the keyword sequence `PROCEDURE __INLINE__`. Again this matches the GNU C compiler. If a procedure is exported then the procedure is still inlined locally and a copy of the code is placed in the object file to resolve external references.

The `__BUILTIN__` keyword occasionally appears in a definition module. In this case the procedure is implemented internally within the compiler back end. This allows the compiler to utilise its library of optimal routines without changing an applications import list. For example `memcpy`, `alloca` can be exported from the library `libc` and `sin`, `cos`, `log2` are exported from `MathLib0`.

Figure 1 contains the complete PIM compliant `MathLib0` definition module which defines many standard mathematical functions and two constants.

By examining the definition module the user can immediately determine which of the functions will be in-lined if the appropriate optimisation flags are present on the `gm2` command line. In the implementation module the mapping between PIM function names and the GCC back end functions are stated. Figure 2 contains the first 30 lines of the `MathLib0` implementation module.

In this implementation module the keywords `__ATTRIBUTE__` `__BUILTIN__` are used to denote the mapping between the internal GCC function name and the equivalent Modula-2 function. It is also worth noting that this module imports `cbuiltin` and `libm`. The module `cbuiltin` declares all GCC built-in functions whereas the module `libm` provides access to the C library `libm.a`. The above implementation module is constructed by calling upon `cbuiltin` functions wherever possible and only falling back upon the services of `libm` when no built-in is available.

```
DEFINITION MODULE MathLib0 ;

CONST

pi  =3.14159265358979323846264338327950288419 72;
exp1=2.71828182845904523536028747135266249775 72;

PROCEDURE __BUILTIN__ sqrt (x: REAL) : REAL ;

PROCEDURE __BUILTIN__ sqrtl (x: LONGREAL) :
                      LONGREAL ;

PROCEDURE __BUILTIN__ sqrts (x: SHORTREAL) :
                      SHORTREAL ;

PROCEDURE exp (x: REAL) : REAL ;
PROCEDURE exps (x: SHORTREAL) : SHORTREAL ;

PROCEDURE ln (x: REAL) : REAL ;
PROCEDURE lns (x: SHORTREAL) : SHORTREAL ;

PROCEDURE __BUILTIN__ sin (x: REAL) : REAL ;

PROCEDURE __BUILTIN__ sinl (x: LONGREAL) :
                      LONGREAL ;

PROCEDURE __BUILTIN__ sins (x: SHORTREAL) :
                      SHORTREAL ;

PROCEDURE __BUILTIN__ cos (x: REAL) : REAL ;
PROCEDURE __BUILTIN__ cosl (x: LONGREAL) :
                      LONGREAL ;
PROCEDURE __BUILTIN__ coss (x: SHORTREAL) :
                      SHORTREAL ;

PROCEDURE tan (x: REAL) : REAL ;
PROCEDURE tans (x: SHORTREAL) : SHORTREAL ;

PROCEDURE arctan (x: REAL) : REAL ;
PROCEDURE arctans (x: SHORTREAL) : SHORTREAL ;

PROCEDURE entier (x: REAL) : INTEGER ;
PROCEDURE entiers (x: SHORTREAL) : INTEGER ;

END MathLib0.
```

Figure 1: PIM compliant Mathlib0 definition module

In keeping with other GNU compilers `gm2` allows in-line assembly language statements through the `ASM VOLATILE` keywords. The `ASM` statement in `gm2` is an extension to the statement sequence EBNF rule found in the PIM appendices [Wirth1] [Wirth2] [Wirth3]. These follow the method outlined in the GCC manual [Stallman1]. For example on the Pentium[Intel2] the following function adds the two `CARDINAL`s `i` and `j` together and places the result in `k`.

```
ASM VOLATILE ("movl %1,%%eax; \
               addl %2,%%eax; movl %%eax,%0"
      : "=g" (k)            (* outputs *)
      : "g" (i), "g" (j)    (* inputs  *)
      : "eax") ;            (* we trash *)
```

The `VOLATILE` keyword indicates that the instruction has important side effects and the back end is told not to reschedule other instructions across it. The `"g"` informs the back end that the following expression requires an integer register (`"f"` indicates that a floating point register is required). The output integer variable `k` must have an operand string `"=g"`. Finally the back end is told that the `eax` register is destroyed.

Interfacing to other languages is performed by using a language specific definition. The keywords `DEFINITION MODULE FOR "C"` indicate

```
IMPLEMENTATION MODULE MathLib0 ;

IMPORT cbuiltin, libm ;

PROCEDURE __ATTRIBUTE__ __BUILTIN__
          ((__builtin_sqrt))
          sqrt (x: REAL): REAL;
BEGIN
   RETURN cbuiltin.sqrt (x)
END sqrt ;

PROCEDURE __ATTRIBUTE__ __BUILTIN__
          ((__builtin_sqrtl))
          sqrtl (x: LONGREAL): LONGREAL;
BEGIN
   RETURN cbuiltin.sqrtl (x)
END sqrtl ;

PROCEDURE __ATTRIBUTE__ __BUILTIN__
          ((__builtin_sqrts))
          sqrts (x: SHORTREAL) : SHORTREAL ;
BEGIN
   RETURN cbuiltin.sqrtf (x)
END sqrts ;

PROCEDURE exp (x: REAL) : REAL ;
BEGIN
   RETURN libm.exp (x)
END exp ;
```

Figure 2: Section of Mathlib0 implementation module

the implementation module is written in C. It also causes parameters in all exported procedures to be adjusted to match the C calling convention. The example below shows how access to the `libc` function `printf` is achieved. The first parameter `a: ARRAY OF CHAR` will be mapped onto `char *` but will be type compatible with `ARRAY OF CHAR`, all subsequent arguments will be promoted to the Modula-2 type `SYSTEM.WORD`.

```
DEFINITION MODULE FOR "C" libc ;

EXPORT UNQUALIFIED printf ;

PROCEDURE printf (a: ARRAY OF CHAR; ...) ;

END libc.
```

5 Structure of the GNU C compiler

The internal details of the latest release of the GNU C compiler are well documented in [Stallman1] and older versions in [Pizka] and [Granlund].

Until the introduction of GIMPLE and GENERIC into GCC 4.0 the C compiler could be considered as a one pass compiler with four phases. The first phase parses input source and builds a tree structure describing the program's behaviour. This is then manipulated by the second phase to produce a register transfer language (RTL) description. The RTL is a lisp like generic assembly language and this is heavily optimised by the third phase before being transformed into target assembly language by the fourth phase.

It is the duty of the first phase, the compiler front end, to resolve all types, check the correctness of the declarations and enforce the language rules.

5.1 GNU Modula-2 configuration compliance with the GNU C compiler

The file structure of a GCC front end is expected to contain certain key configuration files together with the source code. GNU Modula-2 adheres to this structure and it can be grafted onto GCC 3.3.6 in the subdirectory `gcc-3.3.6/gcc`. It includes the following configuration files:

- `Make-lang.in` defines the high level rules for building the front end. Typically these include rules to build the compiler driver, in this case the command line tool `gm2`, and the rules to build the `info` files, support tools and different compiler generations.

- `Makefile.in` is only used by the maintainers to create GM2 release snapshots.

- `lang-options.h` defines the GNU Modula-2 language specific options. These include: range checks, semantic checking options, dialect options, optimisation, library and linking options.

- `config-lang.in` describes the executables which will be built and a list of `Makefile`s which will be automatically created [MacKenzie] by `./configure` in the top level directory.

- `lang-specs.h` defines all the command line options which are legal in the front end. It also determines how and which support tools will be invoked. In GNU Modula-2 the C preprocessor can be invoked by `-Wcpp`. The specialist `gm2` specific linking options are also defined in this file. The `-Wmakeall` command line option will compile the current module and all dependents and perform the final link.

The main data type used in the interface between front end and the GCC back end is the `tree`. Trees are used to represent constants, types, variables, procedures and all statements. They maybe chained together to represent parameter lists, sequences of record fields or an enumerated data type. The `tree` is implemented in C as a pseudo abstract data type. In the implementation of GNU Modula-2 front end (itself written in Modula-2 and C) this type is presented as an abstract data type. There exists a definition module `gccgm2.def` which provides a functional interface to a wide range of `tree` operators. A corresponding `gccgm2.c` implements this specification. This works extremely well in practice as the separation and purpose of the Modula-2 and C components are clear.

6 GNU Modula-2 compiler options

The GNU Modula-2 compiler options are fully documented in the `texinfo` based manual [GM2]. This section describes some of the more interesting compiler options. The compiler provides extensive runtime checking through the `-Wbounds`, `-Wreturn`, `-Wnil`, `-Wcase` options, which: check array bounds, functions execute a `RETURN` statement, pointers do not dereference through `NIL` and all case expression values are tested.

The compiler can be told to compile PIM2, PIM3, PIM4 dialect Modula-2 via the `-Wpim2`, `-Wpim3` and `-Wpim4` switches respectively. The `-Wiso` is only partially complete, and it currently gives access to the ISO SYSTEM module and modifies the library path to include the ISO libraries.

The options `-Wextended-opaque` and `-Wcpp` enable abstract data types to be implemented by any type and preprocess all source code with the C preprocessor respectively.

One expected category of users are students learning to program. The option `-Wstudents` checks for bad programming style and it checks whether variables of the same name are declared in different scopes and whether variables look like keywords [Lewis]. The `-Wpedantic` option forces the compiler to reject nested `WITH` statements referencing the same record type and does not allow multiple imports of the same item from a module. It also checks that: procedure variables are written to before being read; variables are not only written to but read from; variables are declared and used. It also checks to see that `FOR` loop indices are not used outside the end of a loop without being reset.

The `-Wpedantic-param-names` ensures that procedure parameter names are the same in the definition module and in the implementation module counterpart. This is not necessary in ISO or PIM versions of Modula-2, but it can be extremely useful, as long as code is intentionally written in this way.

Lastly the `-funbounded-by-reference` option enables optimisation of unbounded parameters by attempting to pass non `VAR` unbounded parameters by reference. This optimisation avoids the implicit copy inside the callee procedure. GNU Modula-2 will only allow unbounded parameters to be passed by reference if, inside the callee procedure, they are not written to, no address is calculated on the array and it is not passed as a `VAR` parameter. Note that it is possible to write code to break this optimisation, therefore this option should be used carefully. For example it would be possible to take the address of an array, pass the address and the array to a procedure, read from the array in the procedure and write to the location using the address parameter. Due to the dangerous nature of this option it is not enabled when the `-O` option is specified.

7 Structure of the GNU Modula-2 front end

The language Modula-2 allows declarations to occur in any order and GNU Modula-2 allows an abstract data type to be implemented as any type (not restricted to a pointer type). The Modula-2 import and export rules together with the out of order declaration naturally lends itself to using a multi-pass approach to compilation.

The GNU Modula-2 compiler uses a `flex` built lexical analysis phase to build a dynamic buffer for all source tokens. This is then parsed twice to resolve enumerated types, exports, abstract data types and all the forward declarations. It is parsed for a third time to produce quadruple intermediate code. At this point the quadruples are optionally optimised and semantically checked before being converted into `tree`s and passed to the GCC backend.

The front end symbol table contains a `tree` field for each table entry. This is necessary to implement the optimisation phases in the front end and which provide extra knowledge for semantic analysis. At this point constants and literals will have their `tree` fields initialised in the front end symbol table. Once all the checking has been performed the remaining front end symbol are converted into `tree`s. This technique works well as the front end handles all the backward declarations and it is only when the front end symbol table is entirely complete that many of the type `tree`s are created. This makes the interface to the back end simpler and much easier to debug. The interface routines can ignore many of the error nodes which are only created by the back end when the input source is illegal.

8 Bootstrapping GNU Modula-2

The GNU Modula-2 front end source tree includes the Modula-2 source code for the compiler and libraries as well as a modified version of `p2c`. The modifications to `p2c` allow it to translate the Modula-2 front end component into C. The main changes were: to implement the PIM2 dialect of Modula-2, implement `BITSET` set types and abstract data types.

When building GNU Modula-2 natively the `make gm2.paranoid` test may be performed, this proceeds to compile the Modula-2 sources into object form using the previous version of the compiler. These objects and the GCC back end are linked to form a second generation compiler. The second generation of the compiler is used to create a third generation of the compiler and finally all three compilers are requested to produce assembly language files for all Modula-2 sources. These assembly language files are then `diff`ed to check that the compiler is completely stable. This paranoid test has proven very worthwhile during the development cycle. It also gives a high degree of confidence that the second generation of the compiler is behaving in exactly the same way as the first generation of the compiler and therefore it can be debugged using `gdb` against the original source code (rather than the translated intermediate C code of the first generation compiler).

9 Implementing open arrays using trees

The `tree`s that the back end provide are used right at the start of the compilation process. Initially the back end creates key base types such as `integer_type_node`, `char_type_node` and constants of zero and one. The Modula-2

front end continues to create language specific types such as `BOOLEAN` and initialises trees for any built-in functions that the back end offers. GNU Modula-2 obtains a reference to `memcpy` and `alloca` in `gccgm2.c`. [2]

```
tree gm2_memcpy_node
       = builtin_function
         ("__builtin_memcpy",
          memcpy_ftype, BUILT_IN_MEMCPY,
          BUILT_IN_NORMAL, "memcpy");

tree gm2_alloca_node
       = builtin_function
         ("__builtin_alloca",
          alloca_ftype, BUILT_IN_ALLOCA,
          BUILT_IN_NORMAL, "alloca");
```

In many instances the Modula-2 types can be mapped onto the equivalent C data types. However, as with many other languages, there will be specialist data types required which have no direct C equivalent. The most prominent examples in GNU Modula-2 are that of the open array or unbounded array and large sets. The open array mechanism allows programmers to specify an array parameter to a procedure has no fixed limit. The example below is a declaration for a procedure to concatenate two strings (performing $a := a + b$).

```
PROCEDURE concat (VAR a: ARRAY OF CHAR;
                  b: ARRAY OF CHAR) ;
```

GNU Modula-2 creates an internal `unbounded` type which is declared as a `RECORD`

```
unbounded = RECORD
               _arrayAddress: ADDRESS ;
               _arrayHigh   : CARDINAL ;
            END ;
```

A call to `concat` will involve the caller creating two unbounded temporary structures for parameters `a` and `b`. It fills in the fields to an `unbounded` structure with the address of the array and the last legal index. These two structures become the parameters into the procedure `concat`. GNU Modula-2 adopts the policy of callee save and therefore in the example `concat` must make a copy of the non `VAR` parameter (`b`). This is achieved using the following `tree` and it is called from within the front end Modula-2 source.

```
nBytes :=
  mult(add(indirect(add(addr(param),
                        offset(arrayHighField)),
                    getIntegerType()),
           getIntegerOne())
       findSize(arrayType)) ;

addr :=
  indirect(add(addr(param),
               offset(arrayAddressField)),
           getPointerType()) ;

newArray := gccMemCopy(gccAlloca(nBytes),
                       addr,
                       nBytes) ;
```

This mechanism works well and utilises the functional interface to the `tree` data structure provided by the GCC back end. Essentially the single front end primitives `VAR a: ARRAY OF` *data type* and `a: ARRAY OF` *data type* are implemented by considering their C equivalent using non simple data types and in-lining calls to libc and in-lining C statements. Of course GNU Modula-2 does not generate C but rather it generates the same internal `tree`s that the GNU C compiler generates. In turn these `tree`s define the construction and manipulation of open array data structures.

10 Implementing sets using trees

In ISO Modula-2 set types may have more members than bits in a machine word. For example a user may define large set types in the following way.

[2] `alloca_ftype` and `memcpy_ftype` are the prototypes for the respective functions

```
TYPE
   foo = SET OF CHAR ;
   bar = SET OF [-1..01000H] ;
VAR
   a: foo ;
   b: bar ;
BEGIN
   a := {'a', 'c', d', 'z'} ;
   b := {1, 2, 3, 5, 7, 11, 01000H} ;
```

As the GCC back end does not contain a large set basic type GNU Modula-2 was forced with two choices. Either a new basic type would be added to the GCC back end or GNU Modula-2 could manufacture this type in a similar way to that of an open array. It was decided to manufacture multi word set types rather than introduce a new data type for the GCC back end. The advantages of this technique include simplicity and a clearer separation between the GCC releases and the GNU Modula-2 front end releases. The front end implements set comparison and set element testing routines based on single a multi word set types. It attempts to propagate constants by providing `tree` functions which exploit constant operands whenever possible.

In fact this practice has proved sensible as from GCC release 4.0 onwards the basic word sized set type has been removed. In its place GCC now provides GIMPLE and GENERIC which allow front ends to construct language specific types. Thus the changes to the GNU Modula-2 front end when migrating to GCC 4.1 will include recreating a set type via GIMPLE and GENERIC. GNU Modula-2 was in effect using the GIMPLE technique for both open arrays and large set types, thus the changes should be reasonably well isolated.

The GNU Modula-2 front end manufactures the `SET OF CHAR` construct for a 32 bit data word length processor by generating a GCC `tree` representing the record shown in figure 3. To hide this transformation from the user there exist a collection of patches to be applied to the

```
RECORD
  : SET OF [CHR(0)..CHR(WordLength-1)];
  : SET OF [CHR(WordLength)
            ..CHR(2*WordLength-1)];
  : SET OF [CHR(2*WordLength)
            ..CHR(3*WordLength-1)];
  : SET OF [CHR(3*WordLength)
            ..CHR(4*WordLength-1)];
  : SET OF [CHR(4*WordLength)
            ..CHR(5*WordLength-1)];
  : SET OF [CHR(5*WordLength)
            ..CHR(6*WordLength-1)];
  : SET OF [CHR(6*WordLength)
            ..CHR(7*WordLength-1)];
  : SET OF [CHR(7*WordLength)
            ..CHR(8*WordLength-1)];
END ;
```

Figure 3: Record representing SET OF CHAR

Modula-2 components of `gdb`. These patches allow users to print types and display data in a Modula-2 source code representation. The modified `gdb` understands that a structure containing word sized sets with contiguous ranges and `NULL` field names are to be displayed as a single large set. This mechanism is a pragmatic and release early [Raymond] solution and works well for small to medium sized sets, clearly a more compile time scalable solution is required for really large sets.

11 Modula-2 and interrupt priorities

Another important aspect of Modula-2 is that it provides all the necessary primitives to implement a microkernel either through language constructs or system procedures [Wirth4]. The language allows for modules to be specified to run with a specific interrupt mask. This has the effect that any procedure declared within a module will also inherit this interrupt mask. Thus when an exported procedure is invoked it will automatically set the processor interrupt mask to that of its parent module and restore the previous processor interrupt mask before

returning. For example in figure 4, it can be seen that procedure `foo` is declared in the inner module which was specified to operate with an interrupt mask of 7 whereas the outer module was specified to operate with an interrupt mask of 0. When the procedure `foo` is called from the outer module the current interrupt mask is saved and set to 7. After `foo` returns the interrupt mask is restored back to 0 again.

```
MODULE outer[0] ;

   MODULE inner[7] ;
   EXPORT foo ;

   PROCEDURE foo ;
   BEGIN
   END foo ;

   END inner ;

BEGIN
   foo
END outer.
```

Figure 4: Example of a procedure associated with an interrupt mask

11.1 Modula-2 and processes

The `SYSTEM` module as defined by Wirth [Wirth1][Wirth2][Wirth3] provides four procedures which can be used to create a process, context switch between two processes and switch to another process should an interrupt occur. The prototypes for these `SYSTEM` module procedures are given in figure 5. The procedure `NEWPROCESS` instantiates the procedure represented by parameter `p` into a process `new`. Whereas the procedure `TRANSFER` context switches from process `p1` to process `p2`. The procedure `IOTRANSFER` initially context switches from process `first` to `second` however when an interrupt occurs it saves the current processor volatile environment in `second` and then context switches back to the process `first`. The `LISTEN` procedure briefly removes the processor interrupt mask.

```
PROCEDURE NEWPROCESS (p: PROC;
                      a: ADDRESS;
                      n: CARDINAL;
                      VAR new: PROCESS)

PROCEDURE TRANSFER (VAR p1: PROCESS;
                    p2: PROCESS)

PROCEDURE IOTRANSFER (VAR First,
                          Second: PROCESS;
                      InterruptNo: CARDINAL)

PROCEDURE LISTEN
```

Figure 5: Prototypes of the SYSTEM procedures which coordinate process activity

Fortunately the GNU Pthread library contains low level context switching primitives and these allow for a straightforward implementation of `NEWPROCESS` and `TRANSFER`. The procedure `NEWPROCESS` is implemented by calling `pth_uctx_create` and `pth_uctx_make`. These two Pthread primitives create a process context. Each Modula-2 process is represented by a single Pthread context with an associated interrupt priority mask. In GNU Modula-2 the definition for the `PROCESS` type and the interrupt range and the implementation of `NEWPROCESS` is shown in figure 6.

The implementation of `TRANSFER` and `IOTRANSFER` are shown in figures 7 and 8 respectively. There are three categories of interrupts currently implemented in this runtime system: input, output and clock interrupts. The input and output interrupts are generated by providing a mapping to a file descriptor and the clock interrupts are simulated through the use of a relative ordered time ascending list. A module `SysVec` provides procedures to map file descriptors onto simulated interrupt vectors and it also implements an interrupt dispatcher. This dispatcher is called whenever the interrupt mask is altered and the duty of the dispatcher is to build the set parameters and time parameters for `pth_select`. The values to the set parameters are derived from the active interrupt list

which were populated by successive calls to `IncludeVector` via the `IOTRANSFER` procedure. A function `TurnInterrupts` was added as a GNU extension to the module `SYSTEM`. This function modifies the current interrupt mask and returns the previous mask value but it will also call the interrupt dispatcher if the new mask allows more interrupts to become visible. The compiler generates calls to `TurnInterrupts` whenever one procedure is about to call another procedure associated with a different interrupt mask (as in figure 4).

```
PROCESS  = RECORD
              context: ADDRESS ;
              ints   : PRIORITY ;
           END ;
PRIORITY = [0..7] ;

PROCEDURE NEWPROCESS (p: PROC; a: ADDRESS;
                      n: CARDINAL;
                      VAR new: PROCESS) ;
TYPE
   ThreadProcess = PROCEDURE (ADDRESS) ;
VAR
   ctx: ADDRESS ;
   tp : ThreadProcess ;
BEGIN
   localInit ;
   tp := ThreadProcess(p) ;
   IF pth_uctx_create(ADR(ctx))=0
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'unable to create user context')
   END ;
   IF pth_uctx_make(ctx, a, n, NIL, tp, NIL,
                    illegalFinish)=0
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'unable to make user context')
   END ;
   WITH new DO
      context := ctx ;
      ints    := currentIntValue ;
   END
END NEWPROCESS ;
```

Figure 6: Implementation of NEWPROCESS and definition of the type PROCESS

Every call to `IOTRANSFER` results in a new `IOTransferState` being constructed and this is kept on the callers stack so that the context of `first` process can be restored when the interrupt is serviced. The procedure `IOTRANSFER` initially saves the current processes context into

```
PROCEDURE TRANSFER (VAR p1: PROCESS;
                    p2: PROCESS) ;
VAR
   r: INTEGER ;
BEGIN
   localMain(p1) ;
   p1.ints := currentIntValue ;
   currentIntValue := p2.ints ;
   IF p1.context=p2.context
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'switching to the same process')
   END ;
   currentContext := p2.context ;
   IF pth_uctx_switch(p1.context,
                      p2.context)=0
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'unable to context switch')
   END
END TRANSFER ;
```

Figure 7: Implementation of TRANSFER

```
IOTransferState =
   RECORD
      ptrToFirst,
      ptrToSecond: POINTER TO PROCESS ;
      next: POINTER TO IOTransferState
   END ;

PROCEDURE IOTRANSFER (VAR First,
                      Second: PROCESS;
                      InterruptNo: CARDINAL)
VAR
   p: IOTransferState ;
BEGIN
   localMain(First) ;
   WITH p DO
      ptrToFirst  := ADR(First) ;
      ptrToSecond := ADR(Second) ;
      next := AttachVector(InterruptNo,
                           ADR(p))
   END ;
   IncludeVector(InterruptNo) ;
   TRANSFER(First, Second)
END IOTRANSFER ;
```

Figure 8: Implementation of IOTRANSFER

`first` and then restores the context belonging to process `second`. The `IOTransferState` is constructed and initialised appropriately. A pointer to this record, (`q` in figure 9) is created when calling `AttachVector`. The procedure `AttachVector` associates `q` with the particular interrupt number. When the interrupt is serviced the dispatcher context switches back

to process `second` by invoking `TRANSFER` and passing the relevant fields of `q`.

```
TRANSFER(q^.ptrToSecond^, q^.ptrToFirst^)
```

The last `SYSTEM` procedure `LISTEN` simply listens to all pending interrupts briefly before returning. `LISTEN` is easily implemented by a call to the interrupt dispatcher after un-masking all interrupts. The `SYSTEM` module also provides a non standard procedure `ListenLoop` which exhibits the same behaviour as:

```
LOOP
   LISTEN
END
```

except that it allows the interrupt dispatcher to block waiting for an interrupt to occur thus respecting the underlying operating system.

The implementation of Modula-2 processes works well, it only amounts to 1600 lines of code and it allows input, output and time based interrupts to be managed through `IOTRANSFER`. These modules form part of the GNU Modula-2 runtime system and they provide a method whereby microkernel executives originally designed for stand alone systems can be executed under UNIX like operating systems.

12 GNU Modula-2 source code

At the time of writing GNU Modula-2 0.5 has been released. This front end can be grafted onto GCC-3.3.6 source tree and the front end contains patches for GCC and GDB. To make the source code grafting and build process as simple as possible there is a build package which downloads the appropriate GCC, GDB and GM2 releases, applies the various patches

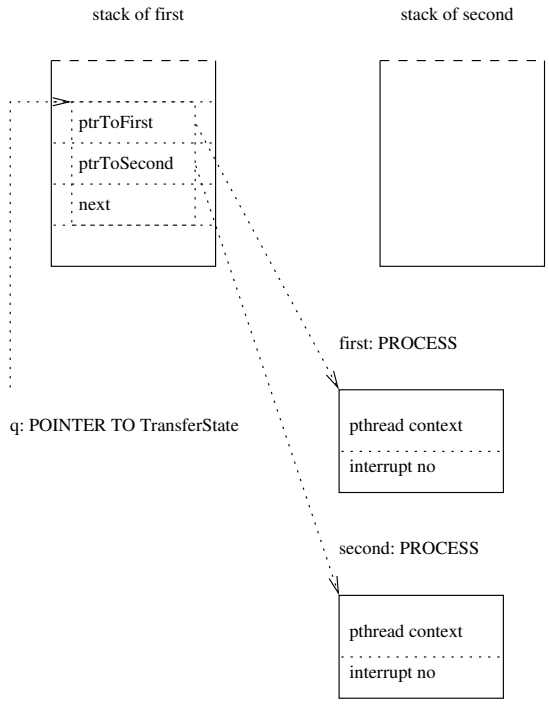

Figure 9: Interaction between IOTransferState and IOTRANSFER

and proceeds to build and install GNU Modula-2 and a modified version of GDB. This utility, `gm2-harness-0.7`, can be found on the GNU Modula-2 web site at `http://www.nongnu.org/gm2`.

13 Conclusions and further work

In conclusion `gm2` has been produced and it builds successfully with the GCC-3.3.6 release on UNIX, GNU/Linux, FreeBSD and Solaris on seven different processors.

The compiler is fully PIM Modula-2 compliant and a complete set of PIM libraries exist. The PIM libraries include: Logitech compatible libraries, University of Ulm libraries and m2f libraries. The user can specify which set of libraries an application should link against.

The compiler will bootstrap reliably and provides accurate debugging information for `gdb`. GNU Modula-2 has also been configured as a cross compiler for the StrongARM [Intel3] and MinGW platforms. It is also the only known free 64 bit implementation of Modula-2 and it will build successfully on the AMD Opteron [Opteron] and Intel Itanium [Intel1] processors running Debian Pure64.

The technique of double book keeping in the symbol table handling has been successful and simplifies the interface between the front and back end. The front end only translates error free and resolved symbols into the GCC `tree` equivalent.

Open array and multi word set implementation in GNU Modula-2 show that GCC front ends can successfully manufacture data types and manipulate data types by providing a mapping onto C derived `tree`s. This will smooth the transition to GCC release 4.1 which uses GENERIC and GIMPLE. It is expected that both: open arrays, set types could be expressed as front end `tree`s which are converted onto back end types appropriately. Ironically the choice of quadruples as front end intermediate code can also be exploited as the quadruples have a direct correspondence with GIMPLE code. This work will be undertaken in the near future.

GNU Modula-2 has implemented the ISO SYSTEM module and some of the ISO language features, clearly however the ISO Modula-2 dialect needs to be completed together with a set of ISO compatible libraries.

Lastly it is a core aim that GNU Modula-2 be integrated within the GCC source tree at a convenient time in the future.

14 Acknowledgements

I would like to thank my employer for funding this research, my colleagues for their support and my family for being so patient. Thank you to all of the readers and contributors of the GNU Modula-2 mailing list for their many valuable bug reports, repeated test builds and patches over the last six years. Many people and organisations have been very generous in providing access to a wide variety of computing equipment which has enabled extensive testing and porting to occur.

Finally a great debt of thanks is owed to the Free Software Foundation and all its contributors without which the source code to GCC would not be free to read, modify and redistribute.

References

[AMD64] AMD, *x86-64 Architecture Programmer's Manual*, AMD USA (2003).

[Bowen] Devon Bowen, *A Highly Portable Modula-2 Compiler*, The State University of New York at Buffalo, Computer Science Department, 226 Bell Hall, Buffalo, New York, 14260, USA (1994).

[Gancarz] Mike Gancarz, *Linux and the Unix Philosophy*, Digital Press (2002).

[GM2] Gaius Mulley, *The GNU Modula-2 front end to GCC*, Edition 0.5, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA (2006).

[Granlund] Torbjörn Granlund and Richard Kenner, *Eliminating branches using a superoptimizer and the GNU C compiler*, ACM SIG-PLAN Notices, Volume 27(7), p341-352 (1992).

[IBM970] IBM, *IBM PowerPC 970FX RISC Microprocessor User's Manual*, IBM, (2006).

[Intel1] Intel, *Intel Itanium Architecture Software Developer's Manual*, Volume 3: Instruction Set Reference, Revision 2.2, Intel (2006).

[Intel2] Intel, *Pentium Processor Family Developer's Manual*, Intel Literature, P.O. Box 7641, Mt. Prospect, IL 60056-7641, USA (1995).

[Intel3] Intel, *Intel StrongARM SA-1110 Microprocessor Developers Manual*, Intel, Intel, USA (2001).

[ISO] ISO/IEC, *Information technology - programming languages - part 1: Modula-2 Language*, ISO/IEC 10514-1 (1996).

[Lewis] Stuart Lewis and Gaius Mulley, *A comparison between novice and experienced compiler users in a learning environment*, 6th Annual Conference on the Teaching of Computing, 3rd Annual Conference on Integrating Technology into Computer Science Education, ITiCSE '98 18th - 31th August, Dublin Ireland, ACM 0-89791-xxx/98/03 (1998).

[MacKenzie] David MacKenzie and Ben Elliston, *Creating Automatic Configuration Scripts*, Edition 2.13, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA (1998).

[Mulley] Gaius Mulley and Keith Verheyden, *Enhancing a Modula-2 compiler to help students learn interactively within the Ceilidh system*, Knowledge Transfer 97 (1997).

[Opteron] AMD, *Software Optimization Guide for the AMD Opteron Processor*, AMD, (2003).

[Pizka] Markus Pizka, *Design And Implementation of the GNU INSEL-Compiler gic*, Technische Universitaet Muenchen, Institut fuer Informatik (1997).

[Pronk] Cornelis Pronk, *Stress Testing of Compilers for Modula-2*, Software Practice and Experience, Volume 22(10), p885-897 (1992).

[Raymond] Eric Raymond, *The Cathedral and the Bazaar*, O'Reilly Publishers, USA (1999).

[Stallman1] Richard Stallman, *Using and Porting the GNU Compiler Collection*, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA (2001).

[Stallman2] Richard Stallman, *GNU Coding Standards*, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA (2006).

[Wirth1] Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin Heidelbery New York, 2nd Edition (1983).

[Wirth2] Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin Heidelbery New York, 3rd Edition (1985).

[Wirth3] Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin Heidelbery New York, 4th Edition (1988).

[Wirth4] Niklaus Wirth, *Design and Implementation of Modula*, Software Practice and Experience, Vol 6(7), p67-84 (1976).