Experiences constructing the GNU Modula-2 front end for GCC *

Gaius Mulley
School of Computing
University of Glamorgan
Treforest, Wales, UK
CF37 1DL
gaius@glam.ac.uk

ABSTRACT

This paper describes a new effort to produce the Modula-2 front end to the GNU C compiler. The strategies used in producing the GNU Modula-2 front end were symbol table double book keeping, post intermediate code semantic checking and a clear separation between front and back end. The result is a highly portable compiler which produces efficient code.

Categories and Subject Descriptors

D.3.11 [**Software**]: Software Engineering

General Terms

Languages, Design

Keywords

Modula-2, GCC, GNU, Compiler

1. INTRODUCTION

The GNU compiler collection (GCC) consists of a number language translators. In the formal `gcc-3.3.2` release these include C, C++, Java, Objective C and Fortran. Additionally there are a number of front ends which are in development and exist outside of this source tree, these include Mercury, Cobol, Pascal, Fortran 95, Ada, Modula-3 and Modula-2. This paper describes the strategy taken to produce `gm2` a Modula-2 front end to the GNU C Compiler.

The motivation for producing `gm2` include providing a migration path for legacy source code. In previous years at Glamorgan we have used Modula-2 while teaching first year programming, compiler design and micro kernel development in the final year. Most of the teaching teams have moved to Java but there is considerable cost associated with translating the teaching compiler and micro kernel into another language. An attractive solution is to provide C and Java language bindings for these applications and keep the legacy Modula-2 code. A compiler which will easily migrate through new hardware platforms and multiple operating systems is highly desirable. Producing a Modula-2 front end onto GCC satisfies these goals and it also has the added advantage that the compiler is freely available for our students and the wider public.

Modula-2 is still used in production systems and has a small but active user community. There is also a considerable amount of interest shown in the concept of `gm2` on the GCC mailing list and also in the `comp.lang.modula2` newsgroup.

There is a lack of commercial Modula-2 compilers which target the new 64 bit microprocessors (AMD64 [1], AMD Opteron [2] and Intel Itanium [8]). A Modula-2 front end for GCC is very pertinent for the Modula-2 community as it provides a migration path for legacy source code written for 16 and 32 bit word computers to be ported to the new generation of 64 bit microprocessors [10][6]. Source code which cannot port to another architecture will die [4].

Modula-2 is also a suitable candidate for modern embedded systems such as the mcf8252 [12] and StrongARM [9]. 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

Previously at Glamorgan a Modula-2 compiler was produced which performed detailed semantic checks and informative error messages [13]. These checks are performed post intermediate code optimisation in an attempt to maximise the knowledge the compiler has about the program source. The code optimisations include dead code elimination, constant folding, copy propagation and common subexpression elimination. To achieve these optimisations the compiler has to build variable life lists which can be exploited to detect elementary infinite loops, manipulation of loop indices and the use of indices outside a `FOR` loop.

There was a previous GNU Modula-2 effort undertaken by the computer science department at the State University of New York at Buffalo [3]. 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

*The GNU Modula-2 mailing list, CVS server and preliminary releases can be found at `http://gnu-modula2.comp.glam.ac.uk`.

BERLIN 2002 Berlin, GERMANY

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.

3. GOALS OF GNU MODULA-2

One of the major questions facing the gm2 team is which Modula-2 dialect should the compiler implement? The two major competitors are the classical *Programming in Modula-2* (PIM) as defined by Wirth [19] [20] [21] and the ISO dialect [18]. The gm2 project has opted for simplicity and a large code base, therefore it will initially support the PIM dialect of the language. The differences between the last three PIM editions are so small that they can all be incorporated into one compiler and they may be selected by a command line option.

ISO compatibility is currently incomplete but a number of ISO features are present, for example large set types and ISO SYSTEM types. As some of the ISO SYSTEM and PIM SYSTEM data types are defined differently the user can specify which standard is required via the -Wiso and -Wpim switches.

Modula-2 allows programmers to declare types, variables, constants and procedures in any order, gm2 must respect this. The compiler must not include any artificial programming limits [15]. The implementation must avoid fixed array sizes and use dynamic data structures throughout.

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 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. One common GNU extension is the ability to in-line assembly language statements using the asm and volatile keywords. Another extension allows local procedures to be in-lined rather than called.

4. EXTENSIONS TO MODULA-2

Modula-2 has no mechanism to manage conditional compilation. While some projects might be able to utilise common definition modules and multiple different implementation modules held in separate directories this can result in a significant duplication of code. For some projects it may be that a cleaner solution is to resort to using the C preprocessor. The C preprocessor can be invoked through the gm2 driver by -Wcpp. This also matches the behaviour of other GNU compiler front ends (namely f77). The -Wcpp option tells the C preprocessor to operate in traditional mode, using assembler as a base language. This 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.

GNU Modula-2 allows local procedures to be compiled inline. 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 in-lined 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.

Below is the completed PIM compliant MathLib0 definition module which defines many standard mathematical functions and two constants.

```
DEFINITION MODULE MathLib0 ;

CONST
   pi   = 3.14159265358979323846264338327950288419716939937510;
   exp1 = 2.71828182845904523536028747135266249775724709369995;

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.
```

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 GCC back end functions are stated. Below is the first 20 lines of the MathLib0 implementation module.

```
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 ;
```

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.

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 [19] [20] [21]. These follow the method outlined in the GCC manual [17]. For example on the Pentium[7] 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 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. AUTOMATIC CREATION OF DEFINITION MODULES FROM C HEADER FILES

This section discusses the `h2def` GNU Modula-2 utility. Since GNU Modula-2 supports interfacing to C using the `DEFINITION MODULE FOR "C"` mechanism it seems sensible to include a tool to translate C header files into these definition modules. The `h2def` utility is provided and includes the following command line options: `-D`, `-C` and `-n`. The `-D`*symbol* option defines a symbol, `-C`*symbol* sets a symbol to be resolved at preprocessing time and `-n`*parameterstem* determines the parameter stem names.

`h2def` will translate all C data types into their Modula-2 equivalent, and map simple `#define` C preprocessing statements into Modula-2 `CONST`s. It will successfully translate pointers to functions into `PROCEDURE` types and, if the `-a` option is present, will map the C prototype *datatype* `*` into `ARRAY OF` *Modula-2 datatype*.

Below is a fragment from the `vga.h` file found in `svgalib` library.

```
#define HAVE_BITBLIT 1
#define HAVE_FILLBLIT 2

typedef struct {
   int width;
   int height;
   int bytesperpixel;
   int colors;
   int linewidth;
   int maxlogicalwidth;
   int startaddressrange;
   int maxpixels;
   int haveblit;
   int flags;
   int chiptype;
   int memory;
   int linewidth_unit;
   char *linear_aperture;
   int aperture_size;
   void (*set_aperture_page) (int page);
   void *extensions;
} vga_modeinfo;

extern
void vga_safety_fork(void (*shutdown_routine) (void));
extern int vga_initf(int);
```

The code below was automatically translated by `h2def`.

```
DEFINITION MODULE FOR "C" vga ;

CONST
   HAVE_BITBLIT  = 1 ;
   HAVE_FILLBLIT = 2 ;
```

```
TYPE
   vga_modeinfo
      = RECORD
           width: INTEGER ;
           height: INTEGER ;
           bytesperpixel: INTEGER ;
           colors: INTEGER ;
           linewidth: INTEGER ;
           maxlogicalwidth: INTEGER ;
           startaddressrange: INTEGER ;
           maxpixels: INTEGER ;
           haveblit: INTEGER ;
           flags: INTEGER ;
           chiptype: INTEGER ;
           memory: INTEGER ;
           linewidth_unit: INTEGER ;
           linear_aperture: POINTER TO CHAR ;
           aperture_size: INTEGER ;
           set_aperture_page:  PROCEDURE (INTEGER) ;
           extensions: ADDRESS ;
        END ;

   t1 = PROCEDURE ;

PROCEDURE vga_safety_fork (shutdown_routine: t1);
PROCEDURE vga_initf (p1: INTEGER) : INTEGER ;

END vga.
```

Currently the limitations of `h2def` are that it cannot translate the C `union` data type, or translate C macros which contain parameters or translate macros which contain C code statements. Nevertheless even with these limitations it greatly improves productivity when generating C definition modules. For example it can translate the GNU/Linux svgalib header file `vga.h` with a minor amount of manual editing. It can also translate the `pthread.h` header file once 10 minutes of user editing has removed the above constraints.

6. STRUCTURE OF THE GNU C COMPILER

The internal details of the GNU C compiler are well documented [17] [14] [5]. [1] Essentially the C compiler is 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 stage before being transformed into target assembly language by the fourth stage.

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

6.1 Integrating a front end into the GNU C compiler

[1]From a practical perspective there exist two tiny front ends which serve to show the directory and file structure of a GCC conforming front end. They also contain a tiny amount of source code which interfaces with GCC to build simple trees. *A Toy Example Language* by Jonathan Bartlett (`http://members.wri.com/johnnyb/compilers`) *Treelang sample language for current snapshot* by Tim Josling (`http://www.geocities.com/timjosling/treelang.diff.txt`).

The file structure of a GCC front end is expected to contain certain key configuration files together with the source code. All front ends exist in the subdirectory `gcc-`*version*`/gcc` and are expected to contain 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` defines how to build the front end and link with the back end. In the context of GNU Modula-2 the goal of this makefile is to produce `cc1gm2`.
- `lang-options.h` defines the front end language specific options which in GCC are normally prefixed by `-W`. For GNU Modula-2 the options include turning on array bound and `NIL` pointer run time checking with `-Wbounds`. Run time checking for functions which finish without executing a `RETURN` statement is enabled with the `-Wreturn` option. Extra compile time semantic checking of programming style is enabled by `-Wstudents`. Every variable name is checked for possible confusion with a similar variable name in a different block and it is checked to see whether it is similar to a keyword. 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. Checks are also performed to make sure procedure variables are initialised before they are read and that variables are both initialised and used. Limited infinite loop detection is performed [13]. Finally it makes sure that a variable acting as a `FOR` loop index is not used outside the loop without being reset.
- `config-lang.in` describes the executables which will be built and a list of `Makefile`s which will be automatically created [11] 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 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) 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.

6.2 Structure of the GNU Modula-2 front end

GNU Modula-2 does not restrict an abstract data type to be implemented as a pointer and it allows declarations to occur in any order. This naturally lends itself to using a multi-pass approach to compilation. Initially a `flex` built lexical analysis builds a buffer for all the 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 intermediate code and this is optimised and then semantically checked.

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 symbols have their `tree` representations created. This technique works well as the front end handles all the backward declarations and it is only once 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.

6.3 Implementing open arrays using `tree`s

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. [3]

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

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

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.

6.4 Implementing sets using `tree`s

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.

```
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. However the GCC back end does provide word size set types. The `gm2` team decided to manufacture multi word set types rather than introduce

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

[3] See appendix A for a definition of `gccMemCopy` and `gccAlloca`

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 GNU Modula-2 front end manufactures the `SET OF CHAR` construct for a 32 bit processor by generating a GCC `tree` representing the following record.

```
RECORD
  w0: SET OF [CHR(0)..CHR(WordLength-1)];
  w1: SET OF [CHR(WordLength)..CHR(2*WordLength-1)];
  w2: SET OF [CHR(2*WordLength)..CHR(3*WordLength-1)];
  w3: SET OF [CHR(3*WordLength)..CHR(4*WordLength-1)];
  w4: SET OF [CHR(4*WordLength)..CHR(5*WordLength-1)];
  w5: SET OF [CHR(5*WordLength)..CHR(6*WordLength-1)];
  w6: SET OF [CHR(6*WordLength)..CHR(7*WordLength-1)];
  w7: SET OF [CHR(7*WordLength)..CHR(8*WordLength-1)];
END ;
```

To hide this transformation from the user there exist a collection of patches to be applied to the Modula-2 components of `gdb`. These patches allow users to print types, display data in a Modula-2 source code representation. `gdb` understands that a structure containing word sized sets with contiguous ranges are to be displayed as a single large set.

This mechanism is a pragmatic and *release early* [16] solution and works well for small to medium sized sets, clearly a more compile time scalable solution is required for really large sets. The scalable solution will be based on a structure containing an array of word sized sets. This work will be undertaken in the near future.

7. CONCLUSIONS AND FURTHER WORK

In conclusion `gm2` has been produced and it builds successfully with the latest GCC release. The compiler is fully PIM Modula-2 compliant and a complete set of PIM libraries exist. 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 [9]. It is also the only known free 64 bit implementation of Modula-2 and it will build sucessfully on the AMD Opteron [2] processor running Debian Pure64 or SuSE 9.1.

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.

Feedback from users indicates that ISO Modula-2 compliance is also desirable and also a complete set of ISO library modules.

More intra language support is also required and perhaps research could be undertaken into providing a mechanism for GNU Modula-2 to link together with some of the Python library modules. Shared library support and an interface to the C++ class construct is also desirable.

8. ACKNOWLEDGMENTS

I would like to thank my employer for funding this research, my colleagues for their support and my family for being so patient. Finally I would like to thank the Free Software Foundation and all the contributors to GCC without which this project could not have happened.

9. REFERENCES

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

[2] AMD. *Software Optimization Guide for the AMD Opteron Processor*. AMD, AMD, USA, 2003.

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

[4] M. Gancarz. *Linux and the Unix Philosophy*. Digital Press, 2002.

[5] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. *ACM SIGPLAN Notices*, 27(7):341–352, July 1992.

[6] J. Hubicka. Porting gcc to the amd64 architecture. In *Proceedings of the GCC Developers Summit*, pages 79–95, May 2003.

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

[8] Intel. *UNIX System V Application Binary interface; IA-64 Processor ABI Supplement*. Intel, Intel, USA, 2000.

[9] Intel. *Intel StrongARM SA-1110 Micrprocessor Developers Manual*. Intel, Intel, USA, 2001.

[10] A. Jaeger. Porting to 64-bit gnu/linux systems. In *Proceedings of the GCC Developers Summit*, pages 107–120, May 2003.

[11] D. MacKenzie and B. Elliston. *Creating Automatic Configuration Scripts*. Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307 USA, 2.13 edition, 1998.

[12] Motorola. *ColdFire Family Programmer's Reference Manual*. Motorola, Motorola, USA, rev 2,07/2001 edition, 2003.

[13] G. Mulley and K. Verheyden. Enhancing a modula-2 compiler to help students learn interactively within the ceilidh system. In *Knowledge Transfer 97*, July 1997.

[14] M. Pizka. Design and implementation of the GNU INSEL-compiler gic. Institutsbericht, Technische Universitaet Muenchen, Institut fuer Informatik.

[15] C. Pronk. Stress testing of compilers for modula-2. *Software Practice and Experience*, 22(10):885–897, October 1992.

[16] E. Raymond. *The Cathedral and the Bazaar*. O'Reilly, 1999.

[17] R. M. Stallman. *Using and Porting the GNU Compiler Collection*. Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307 USA, 2001.

[18] WG13. *Interim Version of 4th Working Draft Modula-2 Standard*. British Standards Institution, 389 Chiswick High Road, working draft 4th edition, 1991.

[19] N. Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin Heidelbery New York, 2nd edition, 1983.

[20] N. Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin Heidelbery New York, 3rd edition, 1985.

[21] N. Wirth. *Programming in Modula-2*. Springer-Verlag,

Berlin Heidelbery New York, 4th edition, 1988.

APPENDIX

A. DEFINITION OF GCCMEMCOPY AND GCCALLOCA

```
/*  gccMemCopy - copy n bytes of memory
 *               efficiently from address
 *               src to dest.
 */

tree gccMemCopy (dest, src, n)
     tree dest, src, n;
{
  tree params
  = chainon(chainon(build_tree_list(NULL_TREE,
                                    convertToPtr(dest)),
                    build_tree_list(NULL_TREE,
                                    convertToPtr(src))),
            build_tree_list(NULL_TREE, n));

  tree functype = TREE_TYPE(gm2_memcpy_node);

  tree funcptr  =  build1(ADDR_EXPR,
                          build_pointer_type(functype),
                          gm2_memcpy_node);

  tree call     = build(CALL_EXPR,
                        ptr_type_node,
                        funcptr, params, NULL_TREE);

  TREE_USED(call)         = TRUE;
  TREE_SIDE_EFFECTS(call) = TRUE ;

  return call;
}
```

```
/*   gccAlloca - given an expression, n, allocate, n,
 *               bytes on the stack for the life
 *               of the current function.
 */

tree gccAlloca (n)
     tree n;
{
  tree params   = listify(n);
  tree functype = TREE_TYPE(gm2_alloca_node);

  tree funcptr  = build1(ADDR_EXPR,
                         build_pointer_type(functype),
                         gm2_alloca_node);

  tree call     = build(CALL_EXPR,
                        ptr_type_node,
                        funcptr,
                        params,
                        NULL_TREE);

  TREE_USED(call)         = TRUE;
  TREE_SIDE_EFFECTS(call) = TRUE ;

  return call;
}
```