GCC .NET—a feasibility study

Jeremy Singer
University of Cambridge Computer Laboratory,
William Gates Building, 15 JJ Thomson Avenue,
Cambridge, CB3 0FD, UK
jeremy.singer@cl.cam.ac.uk

ABSTRACT

We examine the issues involved in producing a back-end for the GNU Compiler Collection (GCC) that targets the .NET Common Language Runtime. We describe a simple back-end that is written in standard GCC style, which interfaces with the register transfer language GCC intermediate representation.

We demonstrate that this simple .NET back-end is able to compile an appreciable subset of the C language. We then consider support for function call handling, object-orientation and language interoperability, amongst other things.

Problems arise because GCC discards much information which could be used to enhance the .NET code generation phase. We briefly describe several possible alternative methods of creating a GCC .NET back-end, which might produce more effective .NET bytecode.

Keywords

Compiler, Back-end, Common Language Runtime

1. INTRODUCTION

In late 2002, Microsoft released the first version of their .NET web services platform. At the heart of this computing environment is the Common Language Runtime (CLR) [2]. The CLR is an object-oriented, type-safe, garbage-collected, secure virtual machine. Many people have claimed that the CLR is a blatant imitation of Sun's Java Virtual Machine (JVM) [9]. There are certainly remarkable similarities between the two. However, while the JVM was explicitly designed to be the target platform for programs written in Java, the CLR is designed to be a language neutral runtime. Thus the CLR provides support for features found in functional programming languages (e.g. tail calls), and in imperative programming languages (e.g. pointer manipulation), as well as all the object-oriented primitive features.

The CLR supports language interoperability. Code written in one .NET language should be able to use data types and routines defined in another .NET language. This feature has huge potential. Programmers are now free to work in their own favourite language, and they can be sure that their code will interoperate with code written in other .NET languages. This also means that there is a unified set of standard .NET libraries, which can be used from any .NET language. This comprehensive set of libraries is invaluable to users of special purpose languages, for whom previously there were few, if any, decent libraries available.

The main .NET compiler is Microsoft's Visual Studio .NET. The major source languages supported are Visual Basic .NET, Managed C++, and C#. We note that Visual Basic .NET and Managed C++ are markedly different from standard Visual Basic and C++. These differences are necessary to allow the languages to target the CLR.

However, there is a wide variety of third-party .NET compilers, for all kinds of programming languages, ranging from old favourites such as COBOL, FORTRAN and Pascal, to weird and wonderful languages such as Haskell, Mercury and Standard ML. The most well-documented .NET compiler is undoubtedly Gough's Component Pascal [4].

All of these .NET compilers transform high-level source code into .NET bytecode, also known as Common Intermediate Language (CIL) [8]. This CIL is the executable code distribution format. The bytecode is translated into native code as required at runtime by the CLR just-in-time (JIT) compiler.

There are currently only CLR implementations for the Microsoft Windows platform, and a Microsoft-sponsored BSD version. Various open-source efforts [17, 16] are underway to reimplement the CLR for Linux. None of these is mature enough for everyday use, so far as we have been able to ascertain. There are also C# compilers, which are not yet complete. There is also one C .NET compiler [16] for Linux, which is also in an unfinished state.

2. BACKGROUND

The GNU Compiler Collection (GCC) [12] is undeniably the world's most popular free compiler. GCC

1st Int. Workshop on C# and .NET Technologies on Algorithms, Computer Graphics, Visualization, Computer Vision and Distributed Computing

Copyright UNION Agency – Science Press
ISBN 80-903100-3-6.

has been the backbone of the open-source movement for the last twenty years. It is arguably the most ubiquitous piece of software in the world, running on everything from mainframes to embedded systems. GCC has been shown to be the best open-source compiler available in terms of quality of executable code produced, in many benchmark tests. It is consistently 20% below the peak performance of Intel's and Microsoft's optimising compilers for x86/IA32, but GCC is as good as, if not better than, everything else in the field (both free and commercial).

GCC is a thoroughly modular compiler. It currently boasts front-ends for C, C++, Objective C, Java and FORTRAN, to name just five main-stream high-level languages. GCC has been ported to all common hardware platforms, and to many more esoteric platforms.

We define *GCC .NET* to mean a modified version of GCC that is able to target the CLR, from its current range of source language front-ends. In this paper, we describe the design and implementation of a standard GCC back-end, which targets the .NET CLR. This is the conventional way of porting GCC to a new hardware platform. As far as we are aware, this is the first ever attempt at creating GCC .NET, as defined above.

The GCC compilation process [13] is shown in figure 1. The source-language-dependent front-end parses high-level source code, and produces a collection of GCC tree data structures. These are compiler abstract syntax trees. GCC then performs various analysis and transformation passes on this tree intermediate representation, which is both independent of the source language and the target platform. Next, GCC generates register-transfer-language (RTL) code from the trees. This is a low-level intermediate representation, which is now target-machine-dependent. GCC does more optimisation passes on the RTL. Finally, the assembly code is produced directly from the RTL by the GCC back-end. This assembly code is then assembled using a standard assembler. (We use the Microsoft .NET IL assembler, ilasm [8].)

The usual way to port GCC to a new platform [13, 10] involves writing a set of "machine description" files, which define how GCC's RTL operations map onto actual target machine instructions. We give examples later, in section 3. There are certain RTL operations which must be defined. However, many are optional, and should only be implemented if the target machine supports them natively. For example, if the target machine has no direct support for a certain arithmetic operation, then GCC will automatically generate RTL code that does not use such an operation, but instead performs an equivalent operation using alternative instructions.

GCC is specifically aimed at CPUs with several 32-bit registers and byte-addressable memory. Most modern instruction set architectures fit this description. However, the .NET CLR is a radical departure from this architectural paradigm. It is a stack-based machine, like the JVM.

egcs-jvm [15] is a port of GCC to the JVM. Waddington experienced many problems attempting to work around the register machine assumptions in the GCC code generator. He has said that it would be easier to throw the JVM away and invent a register-based abstract machine than to try and make GCC generate efficient stack machine code.

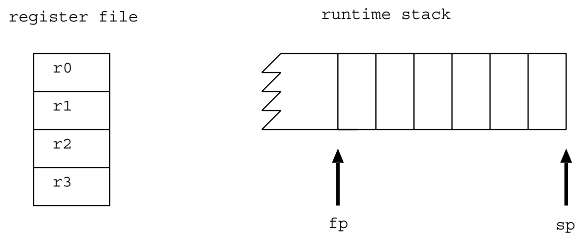

Figure 2: GCC expectations of runtime storage locations

Our GCC back-end is quite similar to egcs-jvm. However, we are able to make good use of the imperative language support features of the CLR, which were not available to Waddington on the JVM. In this way, our code is more efficient and more idiomatic.

3. IMPLEMENTATION

We decided that it would be best to support a simple subset of the C language, in our first attempt at a GCC .NET compiler. We aimed at being able to compile the following features:

- 32-bit integer arithmetic
- indirection and pointer arithmetic
- condition code tests
- intra-procedural jumps and structured control flow
- direct function calls
- runtime stack
- stack allocated integer arrays
- statically allocated integers and integer arrays

We thought that these items would be enough to compile small test programs, from suitable C benchmark suites.

According to Nilsson [10], the first task, when porting GCC, is to clearly define the fundamental machine properties (endianness, number of registers, addressing modes, etc.) and the application binary interface (function calling conventions).

We needed to somehow map the machine properties expected by GCC onto the actual machine properties of the CLR. GCC expects several 32-bit registers available to hold values which are currently in use. GCC also expects a runtime stack which will be used to store local variables, function arguments, and other values which will not fit into registers (spilled values). This is illustrated in figure 2

The CLR does not have any concept of registers. Instead, there is an evaluation stack, which holds values that are currently in use. The evaluation stack is not as flexible as a set of registers, since values can only reside in certain fixed locations if we want to use them at once. To put this another way, we can only

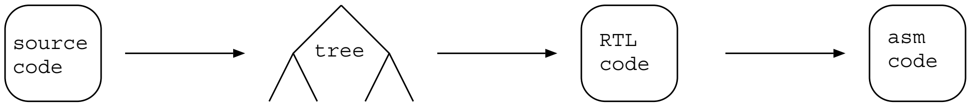

Figure 1: Steps of GCC compilation process

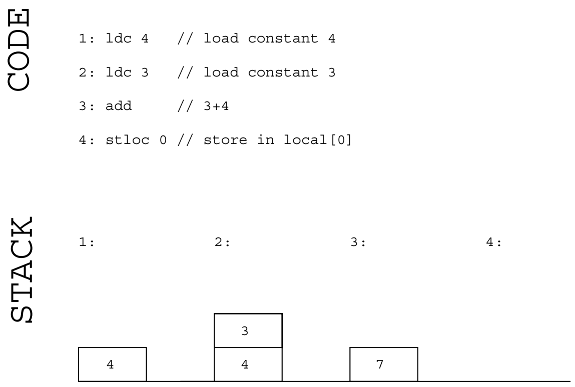
CODE
1: ldc 4 // load constant 4
2: ldc 3 // load constant 3
3: add // 3+4
4: stloc 0 // store in local[0]
STACK
1:
2:
3:
4:
4
3
4
7

Figure 3: Example CLR code, and the evaluation stack at each stage

address values that are on top of the evaluation stack. Figure 3 gives an example of how the CLR execution model works.

The CLR also has an activation record, for the current routine. (This is similar to GCC's concept of a stack frame, but not entirely the same.) A new activation record is created when we enter a new routine. The old activation record is automatically restored when we return from that called routine to the calling routine. Of course, in the case of recursive calls, each activation of a particular routine has its own individual activation record.

The activation record stores two vectors, `args` and `locals`. The `args` vector contains the argument values passed into the routine at the call-site. The `locals` vector contains local variable values and temporary values. The number and types of data in the activation record (both `args` and `locals`) are fixed as part of the routine definition.

We defined the first 32 local variables, `locals[0..31]`, to be 32-bit integer registers, for the purposes of GCC code generation. We specified that two of these registers must be special purpose registers, for stack pointer (`SP`) and frame pointer (`FP`). All the other registers are general-purpose registers, and may be freely used by the GCC register allocator.

We need to support memory references as well. These come for free, since we can treat pointers as 32-bit integers. The CLR will complain about this abuse of its type system, but we can force it to ignore such problems by making our generated code "trusted"—that is, our code is permitted to manipulate pointers without any restrictions.

Below we give the RTL expansion pattern for a simple ADD instruction. We show the the RTL ADD instruction, and its corresponding CLR bytecode. (The machine description files specify how all RTL instructions should be expanded into .NET bytecode.) Notice that we are forced to use the CLR evaluation stack, since this is the only place from where the CLR add instruction can fetch its operands. Notice also that a standard three-address arithmetic operation in RTL normally turns into four CLR instructions.

```
RTL pattern
-----------

ADD r1 <- r2, r3


Corresponding CLR code
----------------------

ldloc r1  // load local var 'r1' onto stack
ldloc r2  // load local var 'r2' onto stack
add
stloc r3  // store the result in 'r3'
```

In general, GCC expects a "condition code" register (`CC`), which stores the result of comparison operations. The `CC` value is checked by conditional branch instructions, to determine whether or not the branch should be taken.

The CLR does not have a "condition code" register. Instead, two values are pushed onto the evaluation stack and then the subsequent conditional branch instruction inspects these two values and decides whether or not to branch. egcs-jvm [15] gets into trouble handling these kinds of instructions, it defines its own special-purpose `CC` register, and checks the value when required. This is not native behaviour for the JVM, but egcs-jvm had no other option.

We handled this case by forcing GCC to place a conditional branch instruction immediately after the corresponding comparison operation. In this way, we can retain the idiomatic .NET compare-and-branch style, as described above, and shown by the RTL to CLR translation below.

```
RTL pattern
-----------

SET cc (CMP r1 r2)   // compare r1 and r2
BEQ label            // if cc==EQ then branch

Corresponding CLR code
----------------------

ldloc r1
ldloc r2
beq label  // branch if top 2 stack items are equal
```

Direct jumps to labels within the same procedure are expected by GCC, and supported by the CLR.

However, GCC also expects some mechanism for indirect jumps, and the CLR, as far as we can see, does not support this. (Nevertheless, indirect function calls are supported.)

Function calling was the most difficult aspect of the implementation. By default, GCC passes some of its parameters in argument registers (if any are defined), and the rest of the parameters are placed on the runtime stack. The standard CLR function calling convention requires the caller to push argument values onto the evaluation stack. These are then transferred into the `args` vector of the called function's activation record.

The simplest method would probably be to constrain GCC to pass all function arguments on the runtime stack, however, this would most definitely be inefficient.

The best method would be to enable GCC to pass all function arguments on the evaluation stack, and these would then appear in the called function's `args` vector. The GCC code generator is not this flexible. We cannot grab all the arguments and tell GCC where to put them. GCC prefers to deal with the arguments itself, before it gets to the function call instruction. So, we cannot tell the difference between ordinary data moves and argument passing, at the code generation phase.

In the end, we decided that we would define eight of our 32 pseudo-registers as outgoing argument registers, and then GCC would use these to store the first eight arguments of a called function. (Any additional arguments are still placed on the runtime stack, but there are not many functions with more than eight arguments.) When we encounter a call instruction, we load the appropriate number of argument registers from the `locals` vector onto the evaluation stack. We then issue a CLR call instruction. This automatically transfers the arguments into the called routine's `args` vector. We do not leave the arguments here. We defined the function prologue code (inserted by GCC at the entry point to every function) to move the argument values from the `args` vector into the appropriate pseudo-registers in the `locals` vector (incoming argument registers). There is a good reason for this. GCC expects all the argument registers to be available for general usage after the argument values are no longer needed. If we pass less than eight parameters, then (unless we supply dummy values for the unused arguments) we will not have eight entries in the `args` vector, so some of the `args` vector will not be addressable. If we call a function with say, three arguments, then only `args[0]`, `args[1]` and `args[2]` are valid. If we then attempt to access `args[3]`, the CLR code will not be valid. It will fail to assemble with the ilasm tool. So, we transfer as many values as there are arguments from the `args` vector into the `locals` vector, and we have arranged that the first 32 entries of the `locals` vector are always addressable.

So, in this way, we have packaged up the CLR calling convention within the GCC function calling mechanism. This means that we should still be able to call other CLR functions which were not written using GCC. (We reconsider this topic in section 5. We represent our parameter passing mechanism diagrammatically in figure 4.

We dealt with function return values in a similar way. The CLR returns values on top of the caller's evaluation stack. GCC expects return values in a register. We defined the function epilogue code to pop the return value from the evaluation stack into the appropriate pseudo-register.

GCC definitely needs a runtime stack. The code generation phase assumes that a runtime stack is available, and makes heavy use of it. (We have noticed that the runtime stack is less intensively used when GCC compiler optimisations are turned on. Then GCC tends to leave as many values in the register file for as long as possible. We might be able to take advantage of this behaviour.) The CLR does not have a runtime stack as such, although the activation record acts like a runtime stack, as we noted earlier. However, we are treating entries in the activation record as our register file.

We create an integer array at the start of each program, and set the `FP` and `SP` registers to point to the end of this array. We have defined the runtime stack to be a descending stack. Since all array indexing is bounds-checked by the CLR, we know that stack overflow (underflow) will be signalled by an `IndexOutOf RangeException` error thrown at runtime. The `FP` and `SP` registers are adjusted as necessary at function calls, whenever the function makes use of the runtime stack. `FP` and `SP` are stored in pseudo-registers (really in the `locals` vector). This means that we don't actually need to save `FP` and `SP` onto the runtime stack, since they are automatically preserved on the caller function's activation record. However, we do have a slight difficulty transferring `FP` and `SP` values to the called function, since we can't access the caller's activation record from within the called function. To overcome this difficulty we could possibly pass `FP` and `SP` as additional hidden arguments to the function, but we decided that this was too ugly. Instead, we write `FP` and `SP` to two global variables just before the function call, and read in `FP` and `SP` just after function entry, if necessary.

Support for stack-allocated integer arrays comes naturally from our implementation of the runtime stack, and from our pointer arithmetic support. GCC knows how to do the rest, and we encountered no problems here.

We treat statically allocated data as static global data fields in .NET. Because of our restriction on types, we are only allowed integers and integer arrays. (Pointers count as integers too, remember.) The CLR expects all such static data to be well-typed, and given initial values. We define static data as value types, which basically means that we are able to do anything we like to the data at these locations. The problem is that value types of different sizes must have different types. We give each item of statically allocated data its own unique type, a subclass of the `ValueType` class. This works fine, and, although it might be inefficient in terms of the amount of metadata (type information) stored, we noticed no considerable problems on our (admittedly small) benchmark tests. We may revisit this area at a later stage. Suffice to say that it works happily at present.

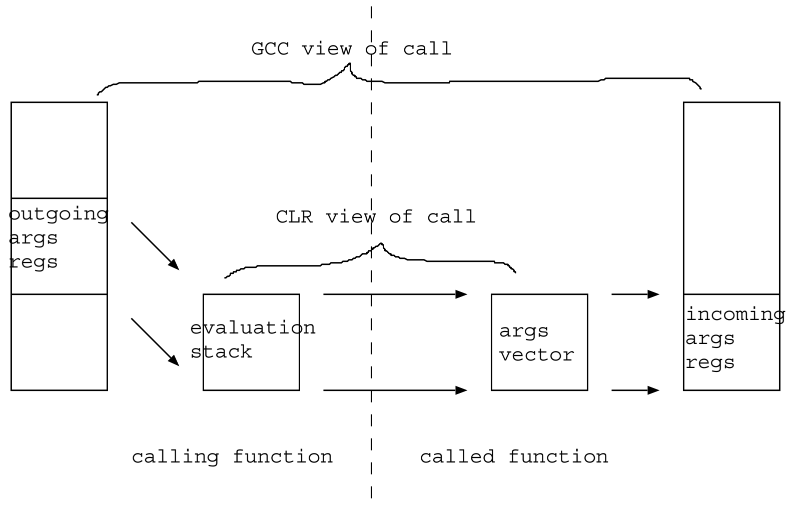

Figure 4: How the CLR calling convention is wrapped up within the GCC calling convention

benchmark	description
ack	ackermann's function
bresenham	line drawing algorithm
mset	mandelbrot calculation
qsort	quick sort
sieve	sieve of eratosthenes
takeuchi	recursive function

Table 1: benchmarks

	size (K)		
benchmark	cl /clr	gcc .net	pnetc
ack	40	4	3
bresenham	40	11	
mset	40	5	3
qsort	36	7	3
sieve	40	49	3
takeuchi	40	3	

Table 2: Executable file sizes for GCC benchmarks

4. RESULTS

These results are only preliminary, since the GCC .NET compiler is still in a highly experimental state.

We used a set of standard GCC benchmark programs [1] to test our GCC .NET compiler. We only selected the programs that used the limited set of C features which our compiler supports properly. These programs are described in table 1.

We ran the benchmark tests through the standard GCC C preprocessor, then passed the resulting preprocessed C source code to our GCC .NET compiler. The output was a single .NET CIL assembly file, in each case. We ran each assembly language file through a simple Perl script which generated the necessary wrapper code for the Microsoft .NET CIL assembler (ilasm) [8]. We then assembled the benchmarks using ilasm, and obtained the resulting EXE files.

We ran these EXE files on the Microsoft CLR implementation. No Linux CLR was able to run the code, which possibly reflects on the immature state of Linux .NET development.

We give details of execution times and code size below. For the purposes of comparison, we also compiled the benchmark test C source code using Microsoft's .NET C compiler (cl /clr) and the Linux-based Portable .NET C compiler (pnetc). pnetc was unable to compile all the benchmark tests, and none of the pnetc executables would run. We reiterate that Portable .NET is a long way away from release quality code.

We also tried to run GCC .NET with optimisations enabled. However, this failed to produce correct assembly code. We need to spend some more time looking into this problem.

Table 2 gives the sizes of the .NET portable executable files generated by the different C .NET compilers. We are slightly suspicious of these results. The Microsoft cl /clr compiler consistently generates large files. We suspect this may be due to security signatures and similar non-essential data included in the executable modules. We did have access to the ilsize program, which reports on how much of each executable comprises code, data, etc. However, ilsize is a part of the Portable .NET distribution, and it failed to work properly on our code. We are also uncertain of the pnetc results. We were unable to get any of the compiled pnetc code to run, under either Linux or Windows CLRs.

Table 3 gives the execution times of the .NET portable executable files generated by the C .NET compilers. The tests were all carried out using Win2K on a 1.4GHz i686. We had to adapt the benchmark tests to make them run for longer, as they were previously too short to time properly. We added some extra loops to the benchmark tests ack and takeuchi.

5. CHALLENGES

In this section, we give details about possible future directions for our current implementation of GCC .NET. Although Fred Brooks affirmed that the first

benchmark	cl /clr time (ms)	gcc .net time (ms)
ack2	8200	13700
takeuchi2	550	840

Table 3: Execution times for GCC benchmarks

attempt should always be thrown away [3], we feel that there is still more that could be done to this simple GCC .NET back-end before it is abandoned as a hopeless cause. Indeed, if all or most of the following concerns can be addressed in a satisfactory manner, then we confidently predict that GCC .NET will be here to stay!

A .NET bytecode analysis and optimisation tool would be a great asset. At present, our back-end emits long forms of all instructions. For example, `ldloc 1` loads the value of `local[1]` onto the evaluation stack. This instruction occupies three bytes, one byte for the `ldloc` opcode, and two bytes for the immediate constant `1`. However, there is an "abbreviated" version of this instruction, namely `ldloc.1`, which is only a single byte long. This kind of code size optimisation could be done by GCC .NET, but it would be just as easy to post-process the assembly code and compact instructions where possible. This would reduce the size of the .NET bytecode generated, but would have no effect whatsoever on the speed of the program execution.

A more complicated bytecode optimiser might be able to spot occasions when values are unnecessarily stored into pseudo-registers, then loaded out again immediately onto the evaluation stack. Apart from conditional branches and function calls, the evaluation stack is always left empty between GCC RTL instruction expansions. Sometimes, this results in code like:

```
ldloc a
ldloc b
add
stloc a
ldloc a
ldloc c
add
stloc a
```

as the transformation of a high-level statement such as `a = a+b+c`.

An optimal version would look like

```
ldloc a
ldloc b
add
ldloc c
add
stloc a
```

which takes full advantage of the evaluation stack to store the intermediate value `(a+b)`. A clever optimiser might be able to perform transformations like this on the bytecode, which would result in improved code size and speed.

Various Java bytecode optimisers have been developed, which work along these lines, such as BLOAT [11] and Sable [6].

At present, GCC .NET only supports 32-bit integers. Realistically, we need to handle 8-bit bytes as well as floating-point arithmetic. The difficulty is that the CLR is strongly typed, which means that pseudo-registers (really CLR `locals`) declared to hold 32-bit integer values can only hold 32-bit integer values. We would need another set of pseudo-registers to handle 8-bit integer values, and further sets of pseudo-registers for each flavour of floating-point types we care to support. We admit that this is not an impossibility, but it does start to look a little untidy after a while. Then consider the GCC runtime stack. This is also strongly typed, so we would need a separate stack for each primitive datatype. There doesn't seem to be a simpler solution to this problem.

By default, the CLR does garbage-collected memory management. The C programming language is not renowned for its amenity to garbage-collection. The CLR does permit explicit memory management, and we may investigate this avenue further.

GCC is far more than just a C compiler. (In the olden days, we assumed GCC stood for GNU C Compiler, whereas now we are led to believe that GCC stands for GNU Compiler Collection.) There are front-ends for C++, Objective C and Java, which are probably the most popular object-oriented programming languages. However, source code from all these object-oriented languages can be reduced to the standard GCC RTL intermediate form. Therefore, we should be able to compile C++, Objective C and Java into .NET bytecode, if we implement enough of the GCC .NET back-end. We have not conducted any comprehensive tests, although initial investigations have shown that extremely simple object-oriented C++ programs can be compiled by GCC .NET.

This would be a neat way of producing Java .NET, for instance. However, we are unable to make use of the CLR object-oriented primitive instructions, since all of the class information has been eliminated by the RTL stage. Instead, we can only generate ugly .NET bytecode that does all the object-orientation explicitly and in an extremely low-level manner, using indirect function calls and pointer arithmetic. This is bound to be slower and less elegant than using the CLR object-oriented primitives.

In our earlier discussion of parameter passing for function calls, we showed that we only use the CLR `args` vector as temporary storage, and move the argument values into pseudo-registers as soon as possible. This is certainly a workable solution, but not the most elegant.

Variable length argument lists are supported natively by the CLR, with special primitive instructions to handle them. GCC also has special support for varargs. We need to match up the CLR and GCC varargs code.

One of the most compelling points of .NET is its language interoperability features. That is to say, routines written in one language can be called by other routines written in a different language, provided both compilers implement the .NET Common Language Specification. This allows us to make use of the extensive .NET class libraries, as well as third-party .NET code.

At the moment, GCC .NET does not support the Common Language Specification. There is no way of calling external .NET functions from C source code.

We need to address this issue. In our test programs, we manually inserted calls to .NET library routines into the GCC .NET generated CIL assembly code, to perform debugging print statements, and to time program execution. This demonstrated that language interoperability is not impossible, but we need to do lots of work to make it happen automatically within GCC .NET.

Another desirable goal is to be able to call native routines from code executing within the CLR. This is done on Microsoft's .NET C compiler by using the p/invoke (platform invocation) subsystem of the CLR. We need to be able to handle this. pnetc also claims to be able to call native code under Linux, but we have not tested this out yet.

The .NET bytecode currently generated by GCC .NET is unverifiable. That is, it cannot be shown to be safe to execute. This is because we make liberal use of pointer arithmetic, and casting between pointer and integer types. It would be nice to be able to generate verifiable code for source programs that do not do any explicit pointer arithmetic. That would mean eliminating all our unsafe pointer manipulations from the GCC .NET generated code. We currently use unsafe pointers for the runtime stack, and for static global data.

6. ALTERNATIVE APPROACHES

GCC has the facility of using a set of stack-like registers. The i386 (x86/IA32) back-end uses this for the floating point register stack. We could possibly make use of this mechanism to model the CLR evaluation stack. At present, GCC .NET is unaware of the evaluation stack, in between RTL instructions.

Older versions of GCC have been ported to the transputer architecture, which is a stack-based machine. The transputer back-end produces fairly standard pseudo-register machine code and then transforms this into more stack-friendly code. It may be possible to adapt some of the transputer back-end ideas to work on the CLR.

The Portable .NET FAQ [16] suggests that a stack-transfer-language (STL) should be created, to be used instead of RTL for the GCC .NET back-end. This would also be of benefit to GCC Java (GCJ). STL would conceivably make code generation much more straightforward for stack-based machines like the CLR and the JVM.

Of course, the real issue is that GCC has thrown away most of the useful information by the back-end code generation phase. (Object-oriented information, type information, etc. has almost completely disappeared.) Perhaps it would be better to begin transformation to .NET bytecode at an earlier stage of the compilation process. It is difficult to interface to the GCC trees intermediate form. However, a new tool called gccxml [7] has recently been developed. This uses the standard GCC front-ends, produces the GCC parse trees and then dumps these out in XML format. We give a simple example in figure 5.

In this example, lots of information about types and objects is retained, which we could put to good use in our code generation. Perhaps this high level view of the program source code is a better thing than simplistic RTL, and perhaps we should generate .NET bytecode this way. It would require a lot more work to generate code from an XML parse tree, and there is little possibility of optimisation, (but perhaps we leave that until the JIT compiler at runtime), however, the executable bytecode produced could potentially be of far purer quality.

7. RELATED WORK

Waddington [15] ported GCC to the JVM, for subset of the C language (quite similar to our subset given in section 3). egcs-jvm does not appear to be under active development any longer.

The JVM Languages webpage [14] lists over 100 different compilers that target the JVM. Among these is c2j, which apparently compiles C programs into JVM bytecode. We have so far been unable to obtain the source code for this compiler.

LCC is another popular C compiler. It is far simpler than GCC, and does not do many optimisations. It is much more straightforward to write a back-end for LCC. Hanson [5] describes his experience of porting LCC to .NET. LCC .NET supports all of Standard C except `setjmp` and `longjmp` and some uses of pointers to functions without prototypes.

pnetc is a C compiler for Portable .NET, a Linux-based implementation of the CLR. We were unable to get pnetc working properly, and assume that this is because the code is still being developed. As far as we can tell, pnetc is a completely new C compiler, rather than an adaptation of an existing C compiler.

8. CONCLUSIONS

We have shown that GCC .NET is feasible, at least for toy C benchmark programs. There remains a great deal of work to be done before GCC .NET can support the whole of Standard C, not even mentioning the other front-ends. We have given a detailed list of the major problem areas which need to be addressed.

Our preliminary results show that GCC .NET produces executable code with acceptable size and speed. Size is the most important factor for most .NET executables. They need to be small so that they can be transferred quickly across slow networks. Hopefully, concerns over code efficiency will be vanquished by the optimisations performed by the CLR JIT compiler, at runtime.

To summarise, GCC .NET is our contribution to the plethora of exciting Linux implementations of .NET related programs—none of which appears to work perfectly, but all of which have some promise for the future.

9. REFERENCES

[1] GCC Benchmarks. `http://savannah.gnu.org/cgi-bin/viewcvs/gcc/benchmarks/`.

[2] Standard ECMA-335 Common Language Infrastructure. `http://www.ecma.ch/ecma1/STAND/ECMA-335.htm`.

```
int a_function(float f, EmptyClass e)
{
}

// ... becomes ...

<Function id="_3" name="a_function" returns="_5" context="_1" location="f0:4">
<Argument name="f" type="_6"/>
<Argument name="e" type="_4"/>
</Function>
```

Figure 5: gccxml output, for a simple C++ source program extract

[3] Fred Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley, second edition, 1995.

[4] John Gough. *Compiling for the .NET Common Language Runtime*. Prentice Hall, 2002.

[5] David R. Hanson. LCC.NET: Targeting the .NET Common Intermediate Language from Standard C. Technical Report MSR-TR-2002-112, Microsoft Research, Nov 2002. `http://research.microsoft.com/~drh/pubs/msr-tr-2002-112.pdf`.

[6] Laurie J. Hendren. Sable, 2001. `http://www.sable.mcgill.ca`.

[7] Brad King. GCC-XML, the XML output extension to GCC, 2002. `http://www.gccxml.org`.

[8] Serge Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.

[9] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

[10] Hans-Peter Nilsson. Porting GCC for Dunces, 2000. `ftp://ftp.axis.se/pub/users/hp/pgccfd/pgccfd.pdf`.

[11] Nathaniel Nystrom. BLOAT (Bytecode-Level Optimizer and Analysis Tool), 1999. `http://www.cs.purdue.edu/s3/projects/bloat`.

[12] Richard M. Stallman. GNU Compiler Collection. `http://gcc.gnu.org`.

[13] Richard M. Stallman. *GNU Compiler Collection Internals*. Free Software Foundation, 2002. `http://gcc.gnu.org/onlinedocs/gccint`.

[14] Robert Tolksdorf. Programming Languages for the Java Virtual Machine. `http://grunge.cs.tu-berlin.de/vmlanguages.html`.

[15] Trent Waddington. egcs-jvm, 2001. `http://sourceforge.net/projects/egcs-jvm/`.

[16] Rhys Weatherley. Portable .NET, 2002. `http://www.southern-storm.com.au`.

[17] Ximian. Mono, 2002. `http://www.go-mono.com`.