CLI-Based Compilation Flows for the C Language

Erven Rohou, Andrea C. Ornstein, Marco Cornero

▶ **To cite this version:**

Erven Rohou, Andrea C. Ornstein, Marco Cornero. CLI-Based Compilation Flows for the C Language. International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, Jul 2010, Samos, Greece. pp.162-169, 10.1109/ICSAMOS.2010.5642069 . inria-00505640

HAL Id: inria-00505640
https://inria.hal.science/inria-00505640

Submitted on 25 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

CLI-Based Compilation Flows for the C Language

Erven Rohou
INRIA Rennes
Rennes, France
erven.rohou@inria.fr

Andrea C. Ornstein
STMicroelectronics
Castelletto, Italy
andrea.ornstein@st.com

Marco Cornero
ST-Ericsson
Agrate Brianza, Italy
marco.cornero@stericsson.com

***Abstract*—Embedded systems contain a wide variety of processors. Economical and technological factors favor systems made of a combination of diverse but programmable processors. Software has a longer lifetime than the hardware for which it is initially designed. Application portability is thus of utmost importance for the embedded systems industry.**

The Common Language Infrastructure (CLI) is a rich virtualization environment for the execution of applications written in multiple languages. CLI efficiently captures the semantics of unmanaged languages, such as C. We investigate the use of CLI as a deployment format for embedded systems to reconcile apparently contradictory constraints: the need for portability, the need for high performance and the existence of a large base of legacy C code.

In this paper, we motivate our CLI-based compilation environment for C, and its different use scenarios. We then focus on the specific challenges of effectively mapping the C language to CLI, and our proposed solutions. We finally analyze the interactions between the CLI environment and native libraries, which is of primary importance for a practical use of the proposed approach.

I. Introduction

Complex embedded systems provide a wide range of dedicated and demanding functionalities, such as communication, multimedia and user interface. Increasing non-recurring engineering costs of integrated circuits push manufacturers to use a given circuit in several products. This trend makes ASICs less attractive and favors programmable solutions [1]. Given the tight area and power constraints, it is impossible to provide these functions using homogeneous programmable architectures. Rather, they are composed of different subsystems, typically including a host micro-controller running the system software, and a growing number of heterogeneous dedicated processors, such as DSP and/or VLIW. Some studies [2] predict that embedded systems will feature hundreds of cores by 2020.

Each platform provider has its own proprietary solutions and evolutions. This results is the extreme diversity of the embedded market, making software productivity a daunting task. Independent software vendors for embedded systems must deal with all the combinations of target processors, toolchains and operating systems, forcing them to restrict their developments to niche domains, and to deal with code duplication, complex build and validation environments and rigid distribution channels, reducing their productivity and market opportunities to a big extent.

This work was supported in part by the European project *Advanced Compiler Technologies for Embedded Streaming (ACOTES)*, under contract IST-5-034869. All authors were with STMicroelectronics when this work was initiated, INRIA also contributed to further developments.

Software has a longer lifetime than hardware: many applications run on hardware that did not exist at the time they were designed. Hardware binary compatibility comes at a high cost, which embedded systems manufacturers can rarely afford. These industrial trends make application portability of utmost importance.

Our main motivation is to extend the benefits of processor virtualization for embedded systems. In particular, we focus on the processor virtualization aspects [3], as an independent feature, while paying particular attention to performance and to integration aspects with existing native or managed environments. Using a platform-neutral bytecode representation is an opportunity for split-compilation: a first compilation pass translates the source language into bytecode, and a second pass converts the bytecode to native machine code. The first pass can run aggressive analyses and encode their results for the benefit of the second pass [4]. Thanks to this additional information, the second pass can apply in a dynamic environment optimizations that would be otherwise too costly.

Legacy code makes the C language mandatory in embedded software. An additional motivation is the higher performance that programmers can achieve in writing "low-level" C, compared to higher-level languages, managed ones in particular. In addition, C does not require any managed environment, reducing the run-time system to a minimum, with benefits in terms of memory footprint and real-time responsiveness (though possible, a JIT compiler is not strictly necessary).

This paper analyzes the interactions of the C language with the CLI (Common Language Infrastructure) framework. CLI is not a usual processor instruction set. It adds new constraints (evaluation stack, strong typing), but it also opens new opportunities in terms of features (support for unmanaged languages) and optimizations. We address in particular portability issues and interaction with existing native environments.

Section II reviews related work. Section III presents some alternatives to the standard static compilation flow made possible by the design of CLI. We then go into the details of code generation (Section IV), library issues (Section V) and intrinsics and builtins (Section VI). Limits are described in Section VII. We conclude in Section VIII.

II. Related Work

Several solutions address platform virtualization. The most notable ones are Java, LLVM and CLI. Java proposes a partial solution to the above-mentioned problems. It defines a bytecode-based virtual machine and a standard library. Java Micro Edition has been widely accepted in embedded systems to provide additional capabilities, like games for cellphones or TV guides for set-top-boxes. However, programs written in Java remain constrained to the host processor for the non-critical part of the application. The primary goal in the definition of the Java bytecode was to support the Java language features, including its safeness characteristics and managed execution environment. It is not well suited to efficiently support unmanaged languages such as C.

LLVM [5] is a compiler framework that defines a low-level code representation appropriate for program analysis and transformation. The representation is typed and language-independent, but it as at a lower level than CLI, and it is not meant as a deployment format. LLVM provides a C compiler.

CLI is a framework that lets applications written in high-level languages execute on different systems, without re-compilation. It is better known as the base of widespread .NET environment. CLI supports a growing set of languages, managed as well as unmanaged. In contrast to Java and LLVM, CLI is an international standard [6].

We have previously shown that CLI is a convenient intermediate representation not only for code size [7] but also performance [8]. However, a robust C compiler is a prerequisite for the adoption of this format in embedded systems, especially to program the media part of the system.

The DotGNU Portable.NET [9] project developed a CLI compiler for the C language. However, object files have a non-standard format, impossible to use in a multi-vendor environment. The execution of the binaries produced by this compiler also requires the support of reflection, a strong constraint for embedded systems. These aspects are further discussed in the next section.

Lcc is a simple retargetable compiler for Standard C. Hanson [10] describes how he targeted Lcc to CLI. He covered most of the language and explains the reasons for his choices, and the limitations. The port was meant more as an experiment to stress Lcc than to produce a robust compiler.

Singer [11] describes another approach to generate CLI from C, using GCC. His implementation starts from the GCC RTL representation and suffers from the loss of high level information. As the title suggests, this is a feasibility study that can handle only toy benchmarks.

We have presented a more mature port of the GCC compiler [12]. It generates correct code for the C99 standard and is publicly available. The contribution of this paper is three-fold:

1) we present our solutions to code generation challenges;
2) we analyze the interactions between CLI and the native libraries necessarily present on a real system;
3) we show the limits of portability of the bytecode representation.

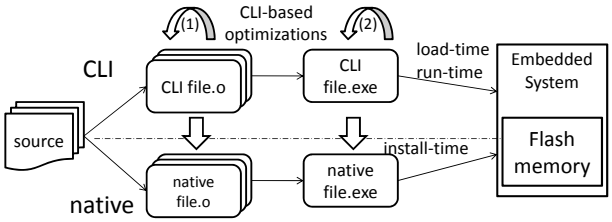

Fig. 1. CLI-based Compilation Scenarios

III. A Customizable CLI-based Compilation Flow

A. Compilation Scenarios

CLI offers several options to introduce flexibility and optimization opportunities at different stages of the compilation flow. Our implementation leaves all options open. Consider Fig.1. At any stage of the compilation process, we have the choice to either produce native binaries, or to keep the code in CLI format, deferring native code generation to a later stage. The decision of which specific scenario is preferable depends on the context, such as distribution format requirements, availability or not of CLI support in the target device, real-time requirements, and possibly others.

This approach also smoothly integrates in existing build systems, such as multi-level Makefiles, by adhering to the traditional separate compilation model and using familiar tools: compiler, assembler, linker...

a) CLI as compiler internal format: as shown in [8], using CLI as a compiler middle-level intermediate representation does not introduce any performance penalty due to loss of information. In particular, we showed that the GCC internal representation can be dumped to CLI and then re-generated in order to proceed with the normal GCC native compilation flow. Besides proving the suitability of CLI to effectively represent the C semantics, this configuration decouples the implementation of compiler middle-level optimization frameworks from a specific compiler internal representation (see (1) in Fig.1).

We use this configuration in the context of MPSoC systems to map software components to processors [13]. We apply coarse-grain transformations to component-based C input specifications, such as component merging and optimization of local communications, directly on the CLI format, rather than using adhoc compiler representations.

b) CLI for link-time, whole program optimization: separate CLI assemblies are linked together to form CLI executables. Once linked, a CLI executable contains all the application modules code in a format that is well suited for further transformation, opening the door to whole program analysis and optimization (see (2) in Fig.1). This is especially convenient when modules are written in different languages.

c) CLI as distribution format: CLI has been defined as a processor-independent distribution format, alleviating the burden of using different toolchains for different target processors. As already highlighted, processor virtualization is particularly welcome in embedded systems, due to the large variety of target processors. In order to exploit CLI as a processor-independent distribution format, the target platform needs to

provide CLI support. In the specific case of our C compilation flow, the generated CLI does not need any virtual machine support for managed features. The required CLI support is reduced to the usual C runtime and possibly a JIT compiler, used in either one of the following configurations on the target devices:

- *install-time:* at application install time, the CLI executable is compiled once and for all into native code, and stored as such in the device permanent (Flash) memory. This is the most JIT-friendly configuration, since the compilation time is visible only during installation.
- *load-time:* the distributed CLI code is kept in CLI format in the device's permanent memory, and translated by the JIT compiler into native code at application load time. In this configuration the whole application is compiled into native at once. The advantage is that, once the application is loaded, it is not impacted by the execution of the JIT compiler anymore, which may be important for real-time applications. The disadvantage is that the whole compilation time is visible by the user at application launch time. Whether it is acceptable or not depends on the type and size of the application.
- *run-time:* this is the classical dynamic JIT configuration used in most managed environment, where application functions are compiled on-demand, depending on the dynamic control flow. In this configuration, the code can also be re-optimized, based on dynamically collected information (e.g. hot spots). The advantage compared to load-time is a better application response time and the potential to achieve higher performance thanks to dynamic optimization. The downside is poor execution time predictability, which can be a problem for real time systems.

The multiple scenarios offered by a neutral bytecode provide a smooth and safe path from pure native, static executables to more dynamic environments. Established industrial flows need not be dramatically modified. Tools can be replaced step-by-step, while the neutral format is introduced. Install-time scenario can later be replaced by a load-time scenario and possibly by a fully dynamic system when needed.

Depending on the specific compiler configuration and runtime environment, different options are available in terms of the generated CLI flavor and the associated toolchain and library interaction requirements. Specifically, in the next subsection, we present some toolchain considerations, while in Section V we analyze the interaction with native and CLI libraries.

B. CLI Toolchain Considerations

When CLI is used only as internal compiler format, there is no need for any specific toolchain support. In this scenario, the CLI is re-injected in the compiler middle-level original internal format, from which normal native object files are generated and then handled by the normal native toolchain. In all the other scenarios, CLI files must be manipulated (i.e. assembled, linked, etc.), and therefore a toolchain support is needed.

CLI does not define any standard format for object files with unresolved references. Instead, it defines the format of CLI *assemblies* for executables and libraries, whose external references, if any, are fully specified (a fully specified reference precisely indicates the external assembly that defines it). This is a problem for representing C object files in CLI because external references in C are not fully resolved (it is the linker's and the loader's task to resolve external references). Static libraries, which are collections of object files, are not defined either.

The lack of CLI object and library formats is a strong limitation for the practical use of a CLI-based C compiler, because many existing C build environments are heavily based on the existence of such formats. In order to overcome this problem, the DotGNU project [9] has defined its own object format and has developed the associated set of tools to support it, i.e. assembler, linker, disassembler, etc.

We adopted a different solution in which object files are represented as standard CLI assemblies, and where unresolved symbols refer to a virtual CLI assembly. Only the linker must be aware of our assumption. This approach lets us use standard CLI tools to manipulate the object files.

Another issue arising from the C language is the initialization of global data, which must occur before the `main` function is called. In a native flow, initialization data is generated by the compiler and stored in a dedicated section (e.g. `.data` and `.bss` in ELF format) of the executable. The content of this section is then copied by the loader into the appropriate memory location before `main` is invoked. This approach is not portable because the layout of global variables may change, depending on the target processor. Initializers must be used, instead of raw data.

Initializers are pieces of code that are executed before `main` is called. The C standard does not specify how this is achieved (§5.1.2 of [14]). The CLI way to implement this is to define a method `.cctor` on the class that requires initialization, possibly the class that contains the `main`. The DotGNU linker generates code that uses the CLI reflection features to collect, at runtime, all the initializers, and to invoke them before starting `main`.

In our implementation for embedded systems, we had to avoid the complexity of supporting reflection. Initializers are merged at link time. The code is inlined and optimized in a single `.cctor` function.

IV. Code Generation

The CLI bytecode is a much higher-level representation than a usual processor instruction set: it retains much of the information present in the programming language (types, symbol names, function signatures) and does not make any assumption on the target resources. By definition, the bytecode is guaranteed to be independent from the actual hardware. Instead of registers, instructions operate on an unbound set of locals (which closely match the concept of local variables) and on elements at the top of an evaluation stack.

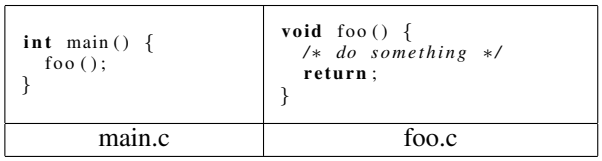

Fig. 2. Calling functions without prototypes

For these reasons, our CLI port is not a back-end in the usual sense of GCC. We kept as much as possible the traditional structure of GCC, but too much of the high-level information is lost at RTL level. We decided to diverge from the usual compilation flow at the end of the middle-end passes and to emit CLI bytecode directly from the GIMPLE representation, skipping all RTL passes. We introduced a new low-level intermediate representation [15] with knowledge of the evaluation stack that enables dedicated program optimizations.

This section presents a number of technical issues we encountered, which are specific to targeting CLI. The first set relates to Standard C [14]. The second one deals with GNU extensions.

A. Standard C

Because the design of CLI is significantly different from a traditional instruction set, it offers a number of new opportunities to the code generator. Additional constraints also derive from the need to strongly type all the manipulated data.

1) CLI stricter than C: In some cases, the code generation is not as straightforward as for native code, because CLI is stricter on types.

- The C90 language [16] lets the programmer call a function even if it has not been declared. The code presented in Fig.2 is correct. The compiler must assume that the returned type is `int`. In the case of the file `main.c`, the value is just ignored. In CLI, though, the compiler has to emit a `pop` instruction to explicitly ignore the returned value placed on the stack. Since, in this example, `foo` does not return any value, the evaluation stack is empty and the `pop` will throw an exception at run time. The earliest time when this problem can be identified is at link time, when all object files are put together. If CLI is used only as an internal format for the compiler, this is a minor issue with no consequence: the code is translated to native before linking. If, instead, the object files are in CLI form and the transition to native is done later on, the CLI linker has to do some extra work to ensure correctness. While the simplest cases can be fixed, the general case is much more complicated.
  To simplify the work of the linker, we chose to support C99, which makes prototypes mandatory (§6.5.2.2 of [17]).
- When passing arguments to a `vararg` function, we pass all integers and pointers of size less than or equal to 32 bits as `unsigned int` and all other integers as `unsigned long`, all floating point values are passed as `double`. Similarly, we extract only `unsigned int`, `unsigned long` and `double` in the implementation of the function. CLI is very strict and a `vararg` value must be extracted with the correct type, otherwise an exception is raised. But in C, it is not an error to extract a pointer as an `int` if they have the same size or to extract an `unsigned int` as a `signed int` or vice versa. To avoid the exceptions we have to emit the correct conversions around the call and after the extraction.

2) CLI higher level than C: Some constructs of CLI make it possible to retain higher-level information than it would be possible with a classical processor instruction set.

- CLI offers a `switch(N)` instruction that implements a jump table. It specifies the branch target for each value of the top of stack in the range [0..N-1]. It is quite compact (one word per branch target, plus five bytes overhead) for dense, zero-based ranges of values. It has the advantage to retain much of the semantics of the C switch statement, without obscuring the control flow graph with tables of labels, or additional basic blocks for the sequences of compare-and-branch instructions. Back-ends can then decide how to implement the switch, based on their own heuristics. We split switches with sparse values into several switches and/or singleton values that are handled separately with a simple if-statement.
- The `setjmp`/`longjmp` pair is one of the trickiest corner cases of the C standard library. It is as close as it gets to the exception handling mechanism of higher level languages. For that reason, it can be implemented in CLI with the exception mechanism. The CLI implementation of `longjmp` just throws an exception of a predefined type. Any occurrence of `setjmp` must be protected with a `try`/`catch` block, and a `leave` statement resets the control flow as needed.
  However, this code generation scheme is only appropriate when the CLI code is meant to be run on an actual virtual machine. If fed to a install-time or load-time compiler, it is very unlikely that the complex `try`/`catch` code pattern will be recognized and emitted as the C programmer expects. Instead, very inefficient code (although correct) is likely to be generated. In this case, it is better to keep calls to builtins that the install-time or load-time compiler must recognize and can handle properly.
- If we are using CLI for portability, there are a few peephole or strength reduction optimizations that we do not want to apply on CLI code. A typical optimizer replaces multiplications and divisions by powers of 2 by the corresponding left or right shifts. In our case, since the generated code is considered an intermediate representation more than actual machine code, it is preferable to keep the more abstract expression. The optimizations might obscure the actual computation to the back-end or to the JIT and inhibit later optimizations, like choosing an appropriate addressing mode. When CLI is used as an internal representation and the final target is known, we want to apply all of the above.

3) Mismatch in concepts: Some concepts of the C language do not have their exact counterpart in CLI, forcing us to express them with other means.

- Even though CLI defines a type *array*, it cannot be used to map C arrays. The reason is that the former are *managed* data, entirely under the control of the garbage collector, and the latter are under the control of the programmer, they are *malloc*'ed and *free*'d memory areas or allocated on the stack with a precise lifespan. We treat C arrays as chunks of memory where accesses are done using pointer arithmetic. All types that end up in an array have to be completely defined, in particular the layout of struct/union types has to be done early to be able to expand the pointer arithmetic.
- The concept of bitfield is not present in CLI. We introduce additional fields and we expand the use of the bitfields with access to the bigger containers and use shift operations. The drawback is that the native layout may differ and marshaling will be needed if such data is passed from CLI to native or vice versa.
- We encode type qualifiers (`const`, `volatile` and `restrict`) in CLI using *custom modifiers* (Partition II §7.1.1 of [6]). This information can be used during the generation of native code to drive optimizations or to generate more accurate debug information. In addition, when a variable is marked as volatile, all accesses are marked with an instruction prefix, as specified by CLI.

4) Portability issues: Portability considerations and the need for reasonable performance also have an impact on the code generation.

- In CLI, we could reference the fields of structs and unions by name and the computation of the layout, unless they are used in an array or they contain bitfields. Their size and alignment must be known at compile time to generate the proper pointer arithmetic needed to access the array. When compiling for a specific target, we can directly use the target rules for size and alignment of types. If, instead, we focus on portability, we use natural alignment. The transition from CLI to native needs marshaling.
- `va_list` is an opaque type in C, but its size and alignment must be known at compilation time, so that it can be used inside structures or in arrays. CLI provides the type `ArgIterator` to handle this language feature, but it is opaque as well. We map `va_list` to a *pointer* to `ArgIterator`: its size and alignment are known at compile time (those of a pointer), and we can statically compute the layout of structures and arrays that contain `va_list` fields.
- For portability reasons, all initializations of local and global variables are expanded. In case of a global variable, we create a function, and we mark it so that the linker recognizes it. In our run-time model, they will all be collected and run before the execution of `main`.
  To avoid code bloat, we optimize the initialization of arrays, structs and unions when the initializer is constant. In such a case, we simulate the initialization offline, and store it in a chunk of memory. At runtime, we only have to do a `memcpy`. For this optimization to be valid, we generate both little-endian and big-endian initializers and choose the correct one at runtime. If the memory images obtained in the two cases are identical, or if we are using CLI in a context where we know the endianness of the target, we emit only one chunk.

B. GNU Extensions

Since our development is a port of GCC, we considered supporting some GNU extensions [18].

- GCC defines attributes on variables, functions and types. There are three main categories:
  1) information to be used only by the front-end,
  2) properties of the object they are attached to,
  3) and those directed to the target (information for the linker, ...)

  The first category does not affect our back-end, since the information they provide has already been consumed. For examples, `__attribute__((unused))` instructs the compiler to ignore unused variables.
  We pass down the information provided by the second category generating CLI attributes in the assembly, so that the second compiler can take advantage of this added information during its optimization passes; a good example of this category is the attribute `pure` or `const` attached to functions.
  Keeping the ones in the third category makes sense only if we know the final target and if we are not compiling for portability; a good example would be the `section` attribute attached to functions, the concept of which section is present on a target is not portable.
- The `asm` keyword lets a developer write inline assembly code in the body of a C function. It can be for performance reason, or to execute an instruction whose semantics is not captured by the C language. Typical examples are instructions to flush the cache, or specific instructions the compiler is not able to exploit. For portability, `asm` expects CLI bytecodes. The difficulty comes from the execution stack. CLI requires that the maximum depth be encoded in the function header. The compiler computes it while emitting code. However, `asm` is opaque to the compiler. The syntax must be extended to express the variation of the evaluation stack depth.
  However, when CLI is used only as an internal representation (the target is known and the developer may even not be aware of the internal use of CLI) the right choice for the implementation of `asm` is to use the native assembler of the target.
- The GNU extension *Labels As Values* lets the user take the address of a label and store it, in order to use it later as the target of a `goto`. CLI does not allow jumps to computed addresses. We associate an ID to each label whose address is taken. The IDs are stored instead of

```
int main(int c,char** v)
{
  void* labels[]=
       {&&l1,&&l2,&&l3};
  int val = atoi(v[1]);
  goto *labels[0];
l1: printf("1\n");
l2: printf("2\n");
l3: return 0;
}
```

```
  call 'atoi' ...
  ldc.i4 4
  mul
  ldloca 'labels?1'
  add
  ldind.i
  switch(?l3,?l2,?l1)
?l1: ...printf("1\n")
?l2: ...printf("2\n")
?l3:
```

Fig. 3. Implementation of the GNU extension *Labels as Values*

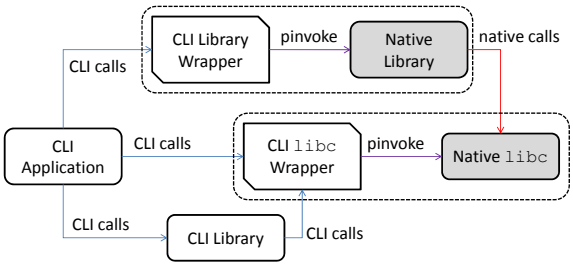

Fig. 4. CLI wrapper around `libc`

the addresses. The `goto` is then replaced by a `switch`. See Fig.3 for an example of code. The major drawback of this implementation is its poor performance, when the programmer probably used it to optimize the code. The main interest of implementing it is to compile legacy code.

- Many cases of nested functions are trivially supported because GCC already rewrites them as standard functions, passing extra parameters when needed. The case when the address of the nested function is taken is trickier, because we need to create a trampoline. A JIT can use the reflection to generate the correct trampoline at runtime.
- GCC also provides a vector extension. The programmer can specify that some types represent packed scalar types. Operations on these types directly map to SIMD instructions when they are available in the instruction set. Vector types and instructions can also be generated by the auto-vectorizer [19]. When emitting CLI code, we map these types and operators to *builtins* proposed by the Mono project in the library *Mono.Simd.dll*. They are then recognized by the Mono JIT which emits efficient native SIMD code. We have shown that this vectorized bytecode is portable and that it runs efficiently, even when the target instruction set does not contain SIMD extensions [20], thanks to the library.

V. Libraries

The bytecode provides only *processor* independence. Achieving *platform* independence is a much wider objective, because it implies also the virtualization of the operating system and a large collection of standard libraries. One of the strengths of Java and the Microsoft .NET environment is indeed the availability of a large set of standard libraries. The CLI standard [6] specifies several libraries, grouped into profiles. Even the smallest one, the Base Class Library, part of the Kernel Profile, uses extensively most of the high-level features of the C# language, requiring therefore a full-fledged CLI virtual machine with support for managed code, reflection, etc. It is not suitable for our lightweight embedded context.

A. General Case

Instead of trying to approach this aspect by providing our own libraries (a large development effort), we decouple the problem by addressing the issues of integrating the code derived from our CLI compilation flow with existing native or managed libraries.

To interact with native code, CLI provides the `pinvoke` primitive to invoke native functions. However, in our context, it is not enough because of data representation issues. Indeed, as mentioned in Section IV, the layout of structs and unions is made explicit in our code generator in order to represent arrays and for enabling all the classical compiler middle-level optimizations on pointer arithmetic in the CLI generation phase. The same representation is kept by the JIT in the generated native code. On the other hand, native libraries use their own data layout, which depends on the specific processor conventions (ABI). These two representations may differ.

There are two approaches for addressing this issue:

- customize the CLI compiler to generate the same data structure layout as the target processor. This limits the portability to processors with identical layouts.
- develop wrappers around libraries to marshal the arguments of `pinvoke` calls, according to the target processor ABI. The advantage of this option is that we do not customize the CLI code to any specific processor ABI, so it remains portable. Only the wrappers need to be developed, making it possible to reuse existing libraries, even in the absence of source code.

In the latter configuration, libraries consist of two parts: a CLI wrapper and the unmodified native library. Because of this split, special care must be taken when the libraries interact, as highlighted in the following subsection.

B. Multi-library Interaction

Libraries interact. When using wrappers, we must pay particular attention to avoid inconsistencies among library invocations. As a general rule, wrappers must be stateless. Consider Fig.4, where the native `libc` is reached both from the native part of the library and from the `libc` wrapper. If part of the state of `libc` resides in the `libc` wrapper, the native library observes an inconsistent state.

Maximum portability is achieved when a complete library is provided in CLI, but similar inconsistencies may arise in case the same library is available also in native form. Again `libc` is a good example: a possible way to offer a portable version of `libc` is to implement it on top of a standardized library layer of the CLI standard, as shown in Fig.5. However, if an application needs also other native libraries in the system, either directly or through wrappers, and if those libraries are dependent on the system `libc`, there is again a risk of inconsistency, since two complete implementations of `libc`

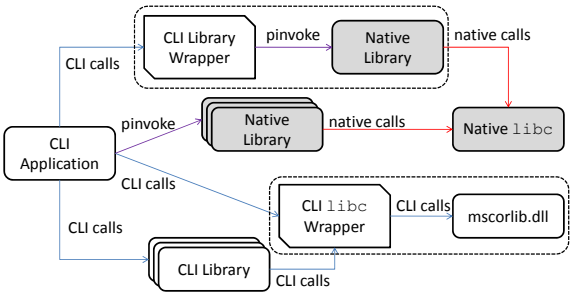

Fig. 5. `libc` implemented using `mscorlib`

would be used concurrently. The only solution to this problem is to make sure that a wrapped native library is always used through its wrapper (no other native library uses it directly).

In conclusion, there is no one-fits-all solution to the problem of libraries. Full-native, native with wrappers, and full-CLI libraries have pros and cons. Portability and performance goals, as well as availability of the library source code, play a role in the choice of the best configuration.

VI. Intrinsics and Builtins

Intrinsics are used by programmers when the compiler is unable to recognize and optimize a critical code pattern. Typical examples are the generation of specific SIMD instructions, or generally accepted mathematical functions. The CLI code generator, however, is special in the sense that it does not know the actual target processor. We end up with two options:

- emit the CLI code sequence that corresponds to the semantics of the given intrinsic.
- emit a CLI call to a function in a support library that implements the semantics of the intrinsic. A naming convention must be agreed upon with the back-end (or the JIT) to make sure that no actual call is generated, but rather the appropriate code sequence.

While this former approach is correct, it misses the whole point of using an intrinsic. The latter relies on the fact that front-end and back-end agree on the names of the intrinsics. But even if the name is not recognized, the emitted native code will still be correct because a library implements the function. Since inlining is a standard optimization, especially for JIT compilers, and the implementation of the intrinsic is small, the function is likely to be inlined, yielding good performance.

Builtins are generated by the compiler itself to carry some interesting information it knows or discovers. GCC, for example, will map a call to `memset()` to `__builtin_memset`.

Some builtins map directly to CLI instructions: for example `__builtin_memset` can be emitted with a `initblk` instruction, `__builtin_memcpy` with `cpblk`.

VII. Limits

While we aim at full portability of our code, some fundamental issues remain. They can be classified in two categories. The first one is related to the C language. The second one is at the border of C and ELF executables.

```
struct node {
  void* data; /* offset 0 */
  struct node* next; /* offset? */
} foo[10];
```

Fig. 6. Array of structs containing a pointer

- CLI defines a type `System.IntPtr` (or `i`), which stands for native integer. Its size is the size of a pointer on the target machine. In other words, it is unknown to the compiler. This is a problem for aggregates (structs or unions) that contains pointers: their size is unknown as well as the offsets of fields located after the pointer. Consider the example of Fig.6. The compiler would not know the size of the structure. An access to `foo[2]` would have to be kept symbolic, as in `*(@foo + 2*sizeof(node))`. This would be legal code, but it would also lead to large and inefficient code, lacking many optimizations like induction variables simplifications. While possible in theory, it would also add extra burden to CLI consumer which may need to apply additional optimizations at run time. The main idea of splitting the compiler in two parts is to have a complex one that goes from C to CLI and a lighter one that translates CLI in native code [4]. Leaving all accesses in symbolic form would achieve exactly the opposite. The first compiler cannot do almost any optimization and all the work has to be done by the second one. We did not consider it realistic in practice.
  We decided to write a CLI generator for a 32-bit machine. A port for a 64-bit machine is obtained by simply changing a parameter in the machine description file.
- Endianness is a key characteristic of a processor. The code produced by our code generator does not depend on it. In some cases, as an optimization, we have to generate two versions with a guard, as explained in Section IV. However, application code that explicitly depends on the endianness, for example thanks to conditional compilation directives (`#ifdef LITTLE_ENDIAN`) cannot be compiled.
- The C language used to produce an ELF shared library allows the library to refer to a global variable defined in the main program. See Fig.7 for an example. This is possible because of the way ELF names symbols, in this case simply the string “x”. Conversely, CLI uses a more precise naming convention: an object is always referenced with its own name and the name of the *assembly* that defines it. In this case, *x* comes from whatever main program links with `libfoo`.
  We believe that this programming style is obsolete and should be avoided. Unfortunately, many legacy projects depend on it. A solution is to use *x* as a reference and to mark it with a special attribute. The linker then has to resolve the symbol by initializing the address of *x* with the proper value. This is how a native linker would handle the situation, using relocations. The induced cost is not different.

```
extern void libfoo(void);
int x; /* global */
int main() {
  x = 2;
  libfoo();
}
```

main.c

```
extern int x;
void libfoo() {
  use(x);
}
```

libfoo.c

Fig. 7. Library referencing a global variable

```
extern int x;
int main() {
  use(x)
   ...
}
```

main.c

```
int x; /* global */
void foo() {
  x = 17;
}
```

libfoo.c

Fig. 8. Main program referencing a variable declared in a library

- A similar, but more frequent, pattern is to define a global variable in a library, to expose a global state to the user, as shown in Fig.8. For example, some implementations of C might define `errno` (§7.5 of [14]) as a global integer variable of `libc`. In a scenario where a native `libc` is used, there is no assembly name, and CLI is missing a feature to access a variable in a native library (only functions can be accessed using `pinvoke`). A solution is to create a native library with getter/setter functions for the variable and call them using `pinvoke`.

VIII. Conclusion

The main purpose of this article is *not* to present performance results or to show any kind of dramatic improvement, but rather to analyze the challenges and opportunities that derive from the compilation of the C language to a CLI framework. A *slight* degradation of performance and/or code size is actually an acceptable price for the advantages brought by virtualization and split-compilation to embedded systems. We implemented our code generator in the GCC compiler version 4.4 [12]. Our experiments show that CLI binaries are 1% larger than native x86 and 12% smaller than SH-4, both quite dense instruction sets. Install-time configuration shows a 2% and 1% average slowdown compared to native x86 and SH-4 respectively. The interested reader can refer to [7] for a discussion of code size and to [8] for performance issues. Usage of builtins and intrinsics, in particular for auto-vectorization is discussed in [20].

This paper illustrates our CLI-based C compilation flow and the solutions to the challenges of efficiently mapping the C language CLI. We address code generation, libraries and compiler intrinsics and builtins. We also show how a standard C compilation flow can be designed using CLI, without breaking the compatibility with existing build environments. Finally, we discuss the limits of the approach.

Our long term goal is to explore solutions for one of the major problems of the embedded market, which is the low software productivity, derived from the extremely high diversity of the target processors. Processor virtualization is necessarily an important element of a possible solution. Our conclusion is that by choosing a well suited processor internal format, such as CLI, it is indeed possible to design a solution based on a highly customizable C toolchain and a lightweight runtime environment, extending the applicability of processor virtualization techniques to the huge C-based software legacy.

References

[1] M. Duranton, S. Yehia, B. De Sutter, K. De Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, N. Navarro, A. Ramirez, O. Temam, and M. Valero, "The HiPEAC vision," Network of Excellence of High Performance and Embedded Architecture and Compilation, Tech. Rep., 2010.

[2] Computing Systems Consultation Meeting, *Research Challenges for Computing Systems – ICT Workprogramme 2009–2010*. Braga, Portugal: European Commission – Information Society and Media, Nov. 2007.

[3] C. Bertin, C. Guillon, and K. De Bosschere, "Compilation and virtualization in the HiPEAC vision," in *Proc. 47th Design Automation Conference (DAC)*, 2010.

[4] A. Cohen and E. Rohou, "Processor virtualization and split compilation for heterogeneous multicore embedded systems," in *Proc. 47th Design Automation Conference (DAC)*, 2010.

[5] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. Intl. Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[6] *International Standard ISO/IEC 23271:2006 – Common Language Infrastructure (CLI), Partitions I to VI*, International Organization for Standardization and International Electrotechnical Commission. Std.

[7] R. Costa and E. Rohou, "Comparing the Size of .NET Applications with Native Code," in *Proc. International Conf. on Hardware/Software Codesign and System Synthesis*, Jersey City, NJ, USA, 2005, pp. 99–104.

[8] M. Cornero, R. Costa, R. Fernández Pascual, A. C. Ornstein, and E. Rohou, "An Experimental Environment Validating the Suitability of CLI as an Effective Deployment Format for Embedded Systems," in *International Conf. on HiPEAC*, vol. 4917, Göteborg, Sweden, Jan. 2008, pp. 130–144.

[9] Portable.NET Project. DotGNU. [Online]. Available: http://dotgnu.org

[10] D. R. Hanson, "Lcc.NET: Targeting the .NET Common Intermediate Language from Standard C," *Software: Practice and Experience*, vol. 34, no. 3, pp. 265–286, 2003.

[11] J. Singer, "GCC .NET – a Feasibility Study," in *1st International Workshop on C# and .NET Technologies on Algorithms, Computer Graphics, Visualization, Distributed and WEB Computing*, Plzeň, Czech Republic, 2003.

[12] R. Costa, A. Ornstein, and E. Rohou, "CLI Back-End in GCC," in *GCC Developers' Summit*, Ottawa, Canada, Jul. 2007, pp. 111–116.

[13] E. Rohou, A. C. Ornstein, A. E. Özcan, and M. Cornero, "Combining Processor Virtualization and Component-Based Engineering in C for Heterogeneous Many-Core Platforms," INRIA, Research Report RR-6933, 2009. [Online]. Available: http://hal.inria.fr/inria-00397823

[14] *International Standard ISO/IEC 9899:TC2 - Programming languages – C*, International Organization for Standardization and International Electrotechnical Commission Std., 1999.

[15] G. Svelto, A. Ornstein, and E. Rohou, "A stack-based internal representation for GCC," in *First International Workshop on GCC Research Opportunities (GROW'09)*, Paphos, Cyprus, Jan. 2009, pp. 37–48.

[16] *International Standard ISO/IEC 9899:1990 - Programming languages – C*, International Organization for Standardization and International Electrotechnical Commission Std., 1990.

[17] WG14, *Rationale for International Standard – Programming Languages – C, Revision 5.10*, Apr. 2003.

[18] R. M. Stallman and the GCC Developer Community, *Using the GNU Compiler Collection*. GNU Press, 2010.

[19] D. Nuzman and R. Henderson, "Multi-platform auto-vectorization," in *Proc. International Symposium on Code Generation and Optimization (CGO'06)*, Washington, DC, USA, 2006, pp. 281–294.

[20] E. Rohou, "Portable and efficient auto-vectorized bytecode: a look at the interaction between static and JIT compilers," in *2nd International Workshop on GCC Research Opportunities (GROW'10)*, Jan. 2010.