An Experimental Environment Validating the Suitability of CLI as an Effective Deployment Format for Embedded Systems*

Marco Cornero, Roberto Costa**, Ricardo Fernández Pascual***, Andrea C. Ornstein, and Erven Rohou

STMicroelectronics
Via Cantonale 16E
6928 Manno
Switzerland

Abstract. Software development productivity for embedded systems is greatly limited by the high fragmentation of platforms and their associated development tools. Platform virtualization environments, like Java and Microsoft .NET, help to alleviate the problem, but because of their advanced run-time features and libraries, they are limited to host functionalities running on the system microcontroller and on top of the operating system. Due to the ever increasing demand for processing power, it is highly desirable to extend the benefits of platform virtualization to the rest of the system programmable resources, media processors in particular, that can boost performance to a great extent. To this aim we are developing a virtualization framework targeting high performance media applications running on deeply embedded media processors, which we combine with traditional host virtualization environments in order to offer a system-wide virtualization solution. In this paper we present an experimental framework based on GCC that allowed us to validate our choice of *CLI* (ISO/ECMA 335 standard, at the base of the better known Microsoft .NET environment) as a suitable processor-independent deployment format for embedded systems. In particular we illustrate the GCC port to *CLI* that we use in our development flow, and we evaluate the quality of the generated bytecode in terms of code size and performance. In order to evaluate performance, we re-inject the generated *CLI* bytecode back into GCC through a GCC *CLI* front-end that we also illustrate, and we complete the compilation process down to native code, whose quality is compared to the code obtained by the normal GCC flow. In so doing we prove that using *CLI* as an intermediate processor-independent deployment format doesn't degrade performance. Compared to other *CLI* solutions, our experimental environment offers a full development flow for the C language, generating a subset of pure *CLI* that does not require any virtual machine support other than a JIT compiler, which is therefore well suited for targeting deeply embedded media processors running high performance real-time media applications.

* This work was partially funded by the HiPEAC Network of Excellence
** This author was working at STMicroelectronics when this work was performed
*** This author performed this work as a HiPEAC PhD student at STMicroelectronics

1 Introduction

The productivity of embedded software development is heavily limited by several factors, the most important being the high fragmentation of the hardware platforms combined with the multiple target operating systems and associated tools. Indeed, in order to reach a wide customer base, software developers need to port, test and maintain their applications to ten's of different configurations, wasting a great deal of effort in the process. Platform virtualization technologies have been very successful in alleviating this problem, as it is evident by the widespread distribution of Java solutions on multiple platforms and operating systems, as well as .NET on Microsoft-supported platforms. However, because of their high-level language support, complemented by sophisticated runtimes and libraries, Java and .NET are well suited only for host functionalities on top of the system microcontroller and operating system. With the growing demand for new and computational intensive applications, such as multimedia, gaming and increasingly sophisticated man-machine interfaces, there is a clear need to extend the computing resources available to software programmers of embedded systems, well beyond the reach of the system microcontrollers. Historically performance scaling has been achieved through increased clock frequency, a path which is not available anymore, especially for portable embedded system, where power consumption is a primary concern. The alternative is multiprocessing, that by the way has been adopted in embedded systems well before its more recent introduction in the PC domain. Multiprocessing comes in many different flavors though, and for efficiency reasons embedded systems have clearly favored highly heterogeneous architectures, typically resulting into two well defined and separated subsystems: the host processing on one side, composed of one microcontroller, possibly scaling to multiple ones in the future, and the deeply embedded side composed of multiple dedicated processors. Because of the difficulty in programming the resulting systems, the deeply embedded side is mostly programmed by the silicon vendors and sometimes by the platform providers, while only the host part is open to Independent Software Vendors (ISV's). In order to respond to the increasing computational power demand, while keeping the required efficiency, it would be highly desirable to grant access to at least part of the deeply embedded computational resources to external ISV's.

Two main issues must be addressed in order to achieve this goal:

1. the programming model and
2. the associated development tools.

Indeed heterogeneous multiprocessor systems are equivalent to small-scale distributed systems, for which a suitable programming model is required. In our research we privilege component-based software engineering practices to tackle this point, but this is not the subject of this paper, so we will not develop it further. For the second point, it is unconceivable to worsen even more the fragmentation problem by introducing new tools for each variant of the deeply embedded processors. That is why we propose to extend the platform virtualization approaches already adopted in environments such as Java and .NET, to

offer a homogeneous software development framework targeting both the host and the deeply embedded subsystems.

Specifically, what we need is a microprocessor-independent format, well suited for software deployment on a wide variety of embedded systems, which can be efficiently Just-In-Time (JIT) compiled for deeply embedded target processors, typically media processors such as DSP's and VLIW's, and which can interact at no additional cost with existing native environments, such native optimized libraries, as well as with existing managed frameworks like Java and .NET. To this aim we have selected a subset of the *CLI* ISO/ECMA 335 standard [8, 14], at the base of the better known Microsoft .NET environment, and the goal of this paper is to illustrate the experimental set up that we have used to validate this choice.

Given our primary target of performance scalability, our most important requirement is efficiency in terms of quality of the final code generated by the JIT, and code size as a second priority. Besides, given the target domain, i.e. compute intensive code on deeply embedded processors, we also constraint ourselves to the traditional programming language used in this context, which is C (we will certainly consider C++ as well in a later phase). And finally, given the real-time nature of our target applications, we avoid for the time being dynamic configurations in which the JIT would be invoked while the application is already running. Instead we pre-compile the input *CLI* code on the target embedded device in one of the following two configurations:

1. at application install-time, *i.e.* the user downloads and application in *CLI* format and during the installation procedure the JIT is invoked to translate the *CLI* into native code, which is then stored into the device persistent memory once and for all; or
2. at load time, *i.e.* the user keeps the application in the device permanent memory in *CLI* format, and each time the application is executed it gets translated by the JIT into native code while being loaded from the persistent to the main memory.

Like in any platform virtualization environment, the compilation process is split in two phases: the generation of the processor-independent format, occurring in the development platform, and the JIT compilation which occurs on the device after the deployment of the applications in a processor-independent format. For the first compilation phase we have chosen the GCC compiler because of its wide adoption and because of its relatively recent introduction of the *GIMPLE* middle-level intermediate internal representation, which we prove in this paper to be well suited for the purpose of generating very efficient *CLI* bytecode. For the JIT part, we are developing our JIT infrastructure targeting our embedded processor family, as well as the ARM microcontroller, with very encouraging results. However the JIT is still work in progress, so we do not illustrate it in this paper. Instead, in order to validate our choice of *CLI*, we have developed an experimental GCC *CLI* front-end, so that we can re-inject the generated *CLI* bytecode back into GCC, and complete the compilation process down to native code, whose quality is compared to the code obtained by the normal GCC flow

(see Figure 1). In so doing we prove that using *CLI* as an intermediate processor-independent deployment format does not degrade performance. We also report the *CLI* code size obtained with our GCC *CLI* generator, which proves to be competitive with respect to native code.

Finally, for what concerns interoperability, *CLI* provides the right primitives (*pinvoke*) for which, once the bytecode is JIT-ted, it can be either statically or dynamically linked with native libraries without any specific restriction or penalty with respect to native code compiled with a normal static flow.

The following section describes our implementation. Section 3 presents our experimental results and analyses. We review some related works in Section 4 before concluding.

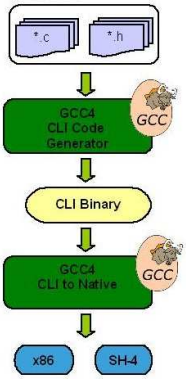


Fig. 1. Experimental Setup

2 Implementation

We implemented our experiments inside the GCC compiler [9] for several reasons: the source code is freely available, it is quite robust, being the outcome of hundreds of developers over fifteen years; it already supports many input languages and target processors; and there is a large community of volunteers and industrials that follow the developments. Its tree-based middle end is also particularly well suited both to produce optimized *CIL* bytecode and to be regenerated starting from *CIL* bytecode.

In the first step, GCC is configured as a *CLI* back-end [5]: it compiles C source code to *CLI*. A differently configured GCC then reads the *CLI* binary and acts as a traditional back-end for the target processor. The *CLI* front-end necessary for this purpose has been developed from scratch for this research [20].

2.1 GCC Structure

The structure of GCC is similar to most portable compilers. A *front-end*, different for each input language, reads the program and translates it into an abstract syntax tree (AST). The representation used by GCC is called *GIMPLE* [19]. Many high-level, target independent, optimizations are applied to the AST (dead code elimination, copy propagation, dead store elimination, vectorization, and many others). The AST is then *lowered* to another representation called register transfer language (RTL), a target dependent representation. RTL is run through low-level optimizations (if-conversion, combining, scheduling, register allocation, etc.) before the assembly code is emitted. Figure 2 depicts this high-level view of the compiler. Note that the parsers of some languages first produce another representation called *GENERIC*, which is then lowered to *GIMPLE* [19]. The GCC internals are described in [4].

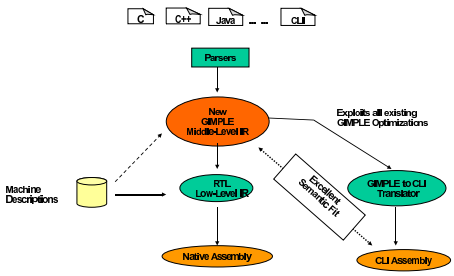


Fig. 2. Structure of the GCC compiler

2.2 *CLI* Code Generator

GCC gets the information about the target from a target machine model. It consists in a machine description which gives an algebraic formula for each of the machine's instructions, in a file of instruction patterns used to map *GIMPLE* to RTL and to generate the assembly and in a set of C macro definitions that specify the characteristic of the target, like the endianness, how registers are used, the definition of the ABI, the usage of the stack, etc.

This is a very clean way to describe the target. But when the compiler needs information that is difficult to express in this fashion, GCC developers have not hesitated to define an ad-hoc parameter to the machine description. The machine description is used throughout RTL passes.

RTL is the lowest level gcc intermediate representation; in RTL

– each instruction is target-specific and it describes its overall effects in terms of register and memory usage;

– registers appearing in RTL instructions may be physical registers as well as virtual, freely inter-mixed (until register allocation pass is run, which leaves no virtual registers);
– memory is represented through an address expression and the size of the accessed memory;
– finally, RTL representation has a very low-level representation of types, which are called *machine modes*.

  Machine modes correspond to the typical machine language representation of types, which only includes the size of the data object and the representation used for it. Examples of RTL machine modes are: BI (single bit), QI (1-byte integer), HI (2-byte integer), SI (4-byte integer), DI (8-byte integer), SF (4-byte floating point number), DF (8-byte floating point number), CC (value of a condition code).

CIL bytecode is much more high-level than a processor machine code. *CIL* is guaranteed by design to be independent from the target machine and to allow effective just-in-time compilation through the execution environment provided by the *Common Language Runtime* (*CLR*). It is a stack-based machine, it is strongly typed and there is no such a concept of registers or of frame stack; instructions operate on an unbound set of locals (which closely match the concept of local variables) and on elements on top of an evaluation stack.

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 much of the high-level information is lost at RTL level, while *CLI* needs to retain it.

Thus, we decided to stop the compilation flow at the end of the middle-end passes and without going through any RTL pass and to emit CIL bytecode from *GIMPLE* representation.

We wrote three specific *CLI* passes:

– CLI *simplification*: most *GIMPLE* tree codes closely match what is representable in *CIL*, this pass expands the ones that do not follow this rule. The output of the pass is still *GIMPLE* but it does not contain tree codes that the emission pass do not handle. The pass can be executed multiple times to avoid putting constraints on other passes. It is idempotent by construction.
– *Removal of temporary variables*: In *GIMPLE*, expressions are broken down into a 3-address form, using temporary variables to hold intermediate values. This pass merges *GIMPLE* expressions to eliminate such temporary variables. Intermediate results are simply left on the evaluation stack. This results in cleaner code and a lower number of locals.
– CLI *emission*: this pass receives a *CIL*-simplified *GIMPLE* form as input and it produces a *CLI* assembly file as output.

With these three passes and a minimal target machine description we are able to support most of C99 [15]. More details of the implementation can be found in [6].

2.3 *CLI* to Native Translation

We have implemented a GCC front-end only for a subset of the *CLI* language, pragmatically dictated by the need to compile the ode produced by our back-end. However, nothing in the design of the front-end forbids us to extend it to a more complete implementation in the future. The supported features include most base *CIL* instructions, direct and indirect calls to static methods, most *CIL* types, structures with explicit layout and size (very frequently used by our back-end), constant initializers, limited access to native platform libraries (*pInvoke*) support and other less frequently used features required by our back-end. On the other hand, unsupported features include object model related functionality, exceptions, garbage collection, reflection and support for *generics*.

Our front-end is analogous to the GNU Compiler for Java (GCJ) [10] which compiles JVM bytecodes to native code (GCJ can also directly compile from Java to native code without using bytecodes). Both front-ends perform the work that is usually done by a JIT compiler in traditional virtual machines. However, unlike JIT compilers, these front-ends are not executed at run-time. Instead, compilation occurs ahead of time. Hence they can use much more time-consuming optimizations to generate better code, and the startup overhead of JIT compilation is eliminated. In particular, our front-end allows compiling *CIL* using the full set of optimizations made available by GCC.

The front-end can compile one or more *CLI* assemblies into an object file. The assembly is loaded and its metadata and code streams are parsed. Instead of writing our own metadata parser, we rely on Mono [3] shared library. Mono provides a comprehensive API to handle the *CLI* metadata and it has been easy to extend where it was lacking some functionality. Hence, the front-end only has to actually parse the *CIL* code stream of the methods to compile.

Once the assembly has been loaded, the front-end builds GCC types for all the *CLI* types declared or referenced in the assembly. To do this, some referenced assemblies may need to be loaded too.

After building GCC types, the front-end parses the *CIL* code stream of each method defined in the assembly in order to build GCC *GENERIC* trees for them. Most *CIL* operations are simple and map naturally to *GENERIC* expressions or to GCC built-in functions. *GENERIC* trees are then translated to the more strict *GIMPLE* representation (*gimplified*) and passed to the next GCC passes to be optimized and translated to native code.

Finally, the front-end creates a main function for the program which performs any required initialization and calls the assembly entry point.

Currently, the main source of limitations in the implementation of a full *CIL* front-end is the lack of a run-time library which is necessary to implement virtual machine services like garbage collection, dynamic class loading and reflection. These services are not required in order to use *CIL* as an intermediate language for compiling C or other traditional languages. Also, *CIL* programs usually require a standard class library which would need to be ported to this environment. The effort to build this infrastructure was outside the scope of our experiment.

2.4 Tools

From the user's point of view, the toolchain is identical to a native toolchain. We essentially use an assembler and a linker. As exepected, the assembler takes the *CLI* in textual form and generates an objet representation. The linker takes several of those object files and produces the final *CLI* executable.

At this point we rely on tools provided by the projects Mono [3] and Portable.NET [1]. We plan to switch to a Mono only solution, to limit the number of dependences we have on other projects, and to avoid using the non-standard file format used by Portable.NET for object files.

3 Experiments and Results

3.1 Setup

This section describes the experimental setup we used to compare the code generated through a traditional compilation flow with the one generated using *CLI* as intermediate representation. GCC 4.1 is the common compilation technology for all the experiments; of course, different compilation flows use different combinations of GCC front-ends and back-ends.

The compilation flows under examination are:

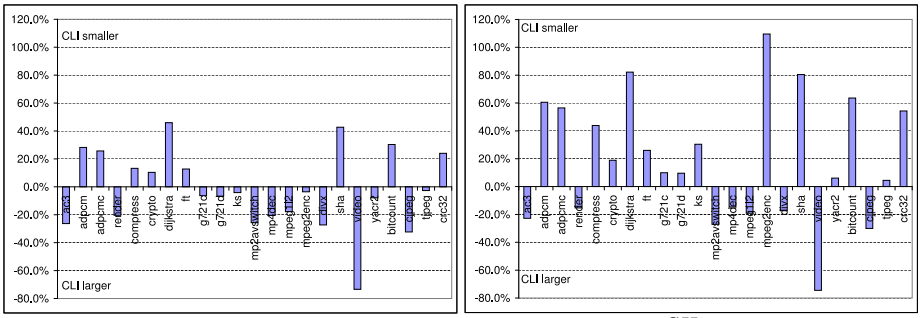
- *configuration a*: C to native, `-O2` optimization level. In other words, this is the traditional compilation flow, it is used as a reference to compare against.
- *configuration b*: C to *CLI*, `-O2` optimization level, followed by *CLI* to native, `-O2` optimization level. This is a compilation flow using *CLI* that always keeps `-O2` as the optimization level. This gives us an upper bound of the achievable performance when going through *CLI*.
- *configuration c*: C to *CLI*, `-O2` optimization level, followed by *CLI* to native, `-O0` optimization level for *GIMPLE* passes and `-O2` for RTL ones. This is still a *CLI*-based compilation flow, in which optimizations at *GIMPLE* level are skipped in the final compilation step. Even though it seems a bit contorted, this setup is important to evaluate up to which degree high-level optimizations can be performed only in the first step. As a matter of fact, in dynamic environments the second compilation step may be replaced by just-in-time compilation, which is typically more time constrained and is likely to apply only target-specific optimizations.

The benchmarks come from several sources: some are MediaBench [17], MiBench [12], others are internally developed. Table 1 gives a short description of each.

We ran our experiments on two targets. The first one is an PC Intel Pentium III clocked 800 MHz, with 256 Mbytes of RAM, running Linux 2.6. The second one is a board developed by STMicroelectronics named STb7100 [24]. The host processor is a *SH-4* clocked at 266 MHz. It features a 64-Mbits flash memory and 64-Mbytes of DDR RAM. The board itself is actually a complete solution single-chip, low-cost HD set-top box decoder for digital TV, digital set-top box or cable box. However, in these experiments we only take advantage of the host processor.

benchmark	description	benchmark	description
ac3	AC3 audio decoder	mp4dec	MPEG4 decoder
adpcm	ADPCM decoder	mpeg1l2	MPEG1 audio layer 2 decoder
adpcmc	ADPCM encoder	mpeg2enc	MPEG2 encoder
render	image rendering pipeline	divx	DivX decoder
compress	Unix compress utility	sha	Secure Hash Algorithm
crypto	DES, RSA and MD5	video	video player
dijkstra	shortest path in network	yacr2	channel routing
ft	minimum spanning tree	bitcount	count bits in integers
g721c	G721 encoder	cjpeg	JPEG encoder
g721d	G721 decoder	tjpeg	optimized for embedded
ks	graph partitioning	crc32	32-bit CRC polynomial
mp2avswitch	MPEG2 intra loop encoder + MPEG1 layer 2 audio encoder		

Table 1. Benchmarks used in our experiments



x86 *SH-4*

Fig. 3. Native code size wrt. *CLI*

3.2 Experiments

To evaluate the relevance of the *CLI* as a deployment format, we ran two experiments. The first one evaluates the size of the code that needs to be shipped on a device: on one hand the *CLI* file, and the other hand the respective native binaries. In Table 2, the second column gives the size of the *CLI* executable. The following two columns give the size of the *x86* binary, in absolute value, and as a percentage of the *CLI*. The respective numbers for *SH-4* are given in the last two columns. Those number are graphically represented on Figure 3.

The second experiment is about performance. Since we break the compilation flow, one might expect that the compiler lose information in the back-end, and thus performance. Table 3 shows the execution times (in seconds, averaged over five runs) of our set of benchmarks compiled with the three compilation flows for *x86* target and *SH-4*.

	CLI	*x86*		*SH-4*	
benchmark	size	size	%	size	%
ac3	86016	63381	-26.3%	66407	-22.8%
adpcm	8704	11160	28.2%	13974	60.5%
adpcmc	8704	10943	25.7%	13621	56.5%
render	144384	114988	-20.4%	122232	-15.3%
compress	16384	18555	13.3%	23567	43.8%
crypto	73216	80796	10.4%	87040	18.9%
dijkstra	7680	11208	45.9%	13990	82.2%
ft	18944	21359	12.7%	23868	26.0%
g721c	19456	18226	-6.3%	21392	10.0%
g721d	19456	18159	-6.7%	21321	9.6%
ks	16896	16196	-4.1%	22034	30.4%
mp2avswitch	272384	202446	-25.7%	198486	-27.1%
mp4dec	67584	53824	-20.4%	57636	-14.7%
mpeg1l2	104960	86279	-17.8%	84863	-19.1%
mpeg2enc	88576	85415	-3.6%	185632	109.6%
divx	67584	49134	-27.3%	55869	-17.3%
sha	7680	10960	42.7%	13858	80.4%
video	1036288	275819	-73.4%	264067	-74.5%
yacr2	37376	34441	-7.9%	39653	6.1%
bitcount	9728	12678	30.3%	15912	63.6%
cjpeg	226304	153161	-32.3%	158330	-30.0%
tjpeg	54272	52826	-2.7%	56682	4.4%
crc32	8704	10794	24.0%	13428	54.3%

Table 2. Code size results for *CLI*, *x86* and *SH-4* (in bytes)

3.3 Analysis

The first comment we make on code size is that, even though *x86* and *SH-4* have quite dense instruction sets, the *CLI* binary is often smaller. The case of *mpeg2enc* on *SH-4* is extreme and comes from the fact that the native compiler decided to statically link part of the math library to take advantage of specialized trigonometric routines. Several benchmarks see their code size increased by the introduction of *SH-4* nops to ensure proper alignment of basic blocks and functions, required to achieve high performance of this architecture.

The worst case comes from the benchmark *video*, where the *CLI* is roughly 74% larger than *x86* or *SH-4*. It turns out that two thirds of the *CLI* code size is made of initializers or arrays of bitfields, for which we have a very poor code generation. A smarter code emission (which we have planned already, but not yet implemented) will get rid of most initializers and generate the values in-place.

Excluding the pathological cases, *video* for both architectures and *mpeg2enc* for *SH-4*, the *SH-4* (resp. *x86*) is 19% (resp. 2%) larger than *CLI*.

There are other opportunities for improvements: in some cases, we have to generate data segments for both little-endian and big-endian architectures. It is

benchmark	*x86*			*SH-4*		
	a	b	c	a	b	c
ac3	0.17	0.18	0.19	0.68	0.71	0.71
adpcm	0.04	0.04	0.04	0.16	0.17	0.17
adpcmc	0.07	0.07	0.06	0.27	0.27	0.27
render	2.15	1.97	2.08	9.87	8.85	8.86
compress	0.03	0.03	0.03	0.50	0.48	0.47
crypto	0.12	0.14	0.15	0.52	0.51	0.53
dijkstra	0.20	0.22	0.22	1.32	1.34	1.34
ft	0.35	0.35	0.29	2.64	2.44	2.62
g721c	0.47	0.46	0.48	1.86	1.66	1.69
g721d	0.43	0.46	0.42	1.54	1.73	1.64
ks	28.00	29.27	30.32	123.44	144.04	140.27
mp2avswitch	2.55	2.55	2.67	12.09	11.55	11.78
mp4dec	0.05	0.06	0.06	0.33	0.35	0.34
mpeg1l2	0.67	0.64	0.63	1.77	1.96	1.95
mpeg2enc	0.76	0.50	0.79	3.37	3.50	4.19
divx	0.39	0.41	0.41	1.27	1.25	1.25
sha	0.30	0.30	0.37	1.98	2.17	2.17
video	0.10	0.10	0.10	0.36	0.37	0.37
yacr2	0.67	0.65	0.70	3.16	3.18	3.08
bitcount	0.03	0.03	0.03	0.13	0.11	0.11
cjpeg	1.72	1.72	1.70	7.73	7.53	7.80
tjpeg	0.48	0.44	0.45	3.05	2.90	3.02
crc32	0.57	0.58	0.55	1.53	1.48	1.52

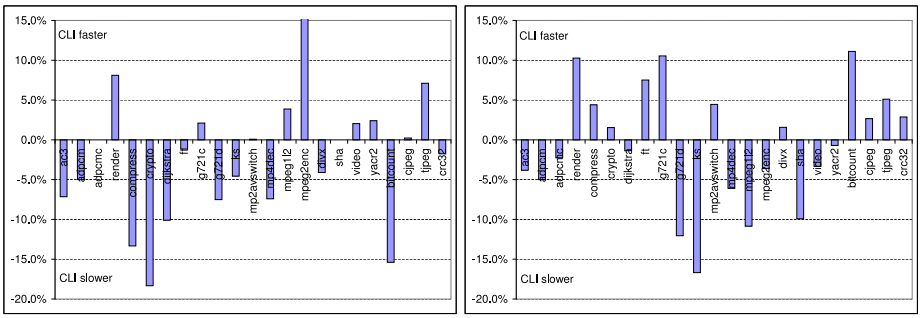
Table 3. Performance results on *x86* and *SH-4* (in seconds)

likely that, at deployment-time, the endianness is known[1]. In this case, the useless data definition could be dropped. Another reduction can come from the fact that *CLI* retains all the source code function and type names in the metadata. In the absence of reflection, which is true for the C language, those names can be changed to much shorter ones. Using only lower case and upper case letters, digits and underscore, one can encode $(2 \times 26 + 10 + 1)^2 = 5329$ names on two characters, drastically reducing the size of the string pool.

Our experiments confirm a previous result [7] that *CLI* is quite compact, similar to *x86* and roughly 20% smaller (taking into account the preceding remarks) than *SH-4*, both notoriously known for having dense instruction sets.

On the performance side, consider the Figure 4 which represents the performance of the binaries generated by the configuration *b* (through *CLI*, at `-O2`) with respect to *a* (classical flow also at `-O2`). It measures the impact on performance of using the intermediate representation. The code generated through *CLI* in configuration *b* is, on average, barely slower than *a*. In other words, using `-O2` optimization level in all cases causes a 1.5% performance degradation

[1] Some platforms are made of processors of both endiannesses. It could be advantageous to migrate the code from one to another and thus to keep both definitions.



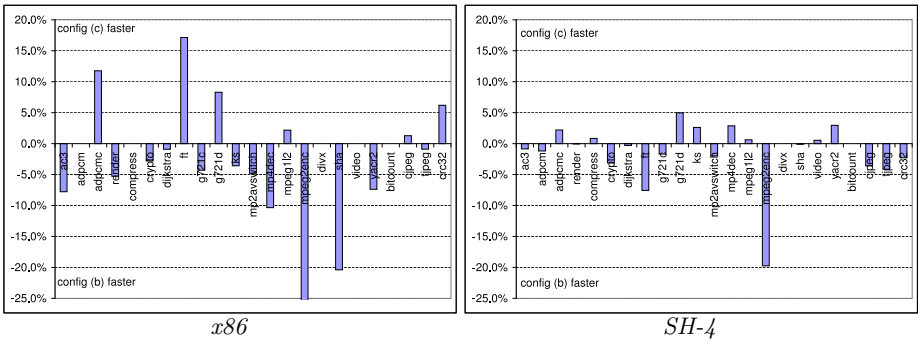
x86 *SH-4*

Fig. 4. Respective performance of *CLI* based code

on *x86* and 0.6% on *SH-4*. The worst degradation is also contained, with -18% for *crypto* on *x86* and -17% for *ks* on *SH-4*.

Understanding the reasons of variations is quite difficult. Even though the compiler infractucture is the same, each target triggers the heuristics in different ways. For example, in the case of *bitcount*, a inner loop has been unrolled by the *SH-4* compiler and not by the *x86* compiler, even though the input file is identical. This lead to the difference in performance seen on Figure 4.

Figure 5 compares the two configurations that use *CLI*, evaluating the need to rerun high-level optimizations in the *CLI* to native translation. The average slowdown is 1.3% on *SH-4* and 4% on *x86*. Excluding the extreme case *mpeg2enc*, they are respectively 0.4% and 1.3%. Most performance degradations are within 5% on the *SH-4* and within 10% on the *x86*. The is a good news because it means that there is little need to run high-level optimizations on the *CLI* executable, it is enough to run the back-end (target specific) optimizer. The poor results of



x86 *SH-4*

Fig. 5. Performance of *CLI* based code without high-level optimizations

some benchmarks are explained by some poor patterns we emit in *CIL*. Array accesses are expressed in *GIMPLE* by a single node, *e.g.* `a[i]`. There is no such abstraction in *CLI* (at least for non managed data). We are forced to emit the code sequence corresponding to `*(@a+sizeof_elem*i)`. When a loop contains several array accesses, we repeat the address computation. This happens at code emission time, and there is no code cleanup afterwards. In configuration *b*, the high level loop optimizer with take care of moving loop invariants and rewriting induction variables. In configuration *c*, this does not happen, leaving the back-end with poor quality code. This is obviously a inefficiency we have to fix.

Keep also in mind that those experiments rely on a prototype implementation of the *CLI* to native code generator. Emphasis has been put on correctness, not yet on quality. Even though the *CLI* generator is in a much better state, we have also identified a number of areas where it could be improved.

4 Related Work

4.1 Not *CLI*-based

In many aspects, the Java framework [18] is the precursor of *CLI*. Similarly to *CLI*, Java defines a bytecode based virtual machine, a standard library and it offers services like garbage collection, multi-threading, etc. Java is widely used in embedded systems in order to provide complementary capabilities, like games for cellphones or TV guides for set-top-boxes. However, it is not adapted for the core software for several reasons.

Java typically incurs a significant performance overhead, not acceptable for performance critical applications; this is mostly because it does not offer install-time compilation, and also because it does not provide the lower-level abstraction the way C does: pointer arithmetic, no garbage collection, no array bound checks, etc. The Java language [11] is high-level and object-oriented language. Porting source code from C to Java typically involves significant changes that may even lead to a full redesign of the application.

AppForge used to be a company selling Crossfire, a plugin for Microsoft Visual Studio .NET that converted the *CLI* bytecode to their own proprietary bytecode. Developers could use VB.NET or C# to develop applications for various mobile devices. The bytecode is interpreted and is not meant to run the performance critical parts of the application, which are still executed natively.

4.2 *CLI*-based

The *CIL* bytecode was first introduced by Microsoft .NET. It has also been standardized by ECMA [8] and ISO [14]. *CIL* has been designed for a large variety of languages, and Microsoft provides compilers for several of them: C++, C#, J#, VB.NET. The C language, though, cannot be compiled to *CIL*.

Mono [3] is an open source project sponsored by Novell. It provides the necessary software to develop and run .NET applications on Linux, Solaris, Mac OS X, Windows, and Unix. It is compatible with compilers for many languages [2].

DotGNU Portable.NET [1] is another implementation of the *CLI*, that includes everything you need to compile and run C# and C applications that use the base class libraries. It supports various processors and operating systems.

Lcc is a simple retargetable compiler for Standard C. In [13], Hanson 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 itself than to produce a robust compiler.

Singer [23] describes another approach to generate *CLI* from C, using GCC. While based on the same compiler, it differs in the implementation: he starts from the 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.

Löwis and Möller [26] developed a *CLI* front-end for GCC with a different goal: while we focus on very optimized C code, they aim at running all the features of *CLI* on the Lego Mindstorm platform [25]. However, they currently support a fraction of the *CLI* features and they can run only small programs.

Very similar to our work is another *CLI* port of GCC done by a student as part of the Google Summer of Code and sponsored by the Mono project [16]. This work is still very preliminary and stopped at the end of the internship.

Some applications may have a long startup time because they are linked against large libraries. Even if the application requires only few symbols, the runtime system might scan a large portion of the library. This also increases the memory footprint off the application. Rabe [21] introduces the concept of *self-contained assembly* to address these problems. He builds a new *CLI* assembly that merges all previous references and does not depend on any other library.

While we priviledged programming embedded systems in C, using *CLI* as an intermediate format, Richter *et al.* [22] proposed to extend *CLI* with attributes to be able to express low-level concepts in C#. They encapsulate in *CLI* classes the notions of direct hardware access, interrupt handler and concurrency.

5 Conclusion

We have illustrated the motivations of our platform virtualization work and the experimental framework that we have used for validating the choice of *CLI* as an effective processor-independent deployment format for embedded systems, in terms of code size and performance. In addition we have described the implementation of our open source *CLI* generator based on GCC4. The presented results show that using *CLI* as an intermediate deployment format does not penalize performance, and that code size is competitive with respect to native code.

We are currently working on optimized JIT compilers for our target embedded processors as well as for the ARM microcontroller. We are also investigating how to optimally balance the static generation of highly effective *CLI* byte-code, complemented with additional information resulting from advanced static analyses, with the dynamic exploitation of that information by our JIT compilers in order to generate highly optimized native code as quickly as possible in the target embedded devices.

References

1. DotGNU project. `http://dotgnu.org`.
2. Mono-compatible compilers. `http://www.mono-project.com/Languages`.
3. The Mono Project. `http://www.mono-project.com`.
4. Richard M. Stallman ad the GCC Developer Community. *GNU Compiler Collection Internals*. Free Software Foundation.
5. Andrea Bona, Roberto Costa, Andrea Ornstein, and Erven Rohou. GCC CLI back-end project. `http://gcc.gnu.org/projects/cli.html`.
6. Roberto Costa, Andrea Ornstein, and Erven Rohou. CLI Back-End in GCC. In *GCC Developers Summit (to be published)*, 2007.
7. Roberto Costa and Erven Rohou. Comparing the Size of .NET Applications with Native Code. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, number ISBN:1-59593-161-9, pages 99–104, Jersey City, NJ, USA, 2005. ACM Press.
8. ECMA International, Rue du Rhône 114, 1204 Geneva, Switzerland. *Common Language Infrastructure (CLI) Partitions I to IV*, 4th edition, June 2006.
9. Free Software Foundation. The GNU Compiler Collection. `http://gcc.gnu.org`.
10. The GNU Compiler for the Java Programming Language. `http://gcc.gnu.org/java`.
11. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, June 2000.
12. Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, USA, December 2001.
13. David R. Hanson. Lcc.NET: Targeting the .NET Common Intermediate Language from Standard C. *Software: Practice and Experience*, 34(3):265–286, 2003.
14. International Organization for Standardization and International Electrotechnical Commission. *International Standard ISO/IEC 23271:2006 - Common Language Infrastructure (CLI), Partitions I to VI*, 2nd edition.
15. International Organization for Standardization and International Electrotechnical Commission. *Internation Standard ISO/IEC 9899:TC2 - Programming languages - C*, 1999.
16. Jeyasankar "Jey" Kottalam. Blog of Jeyasankar "Jey" Kottalam who participated to Google's Summer of Code and did a CIL backend for GCC with Novell. `http://gcc-cil.blogspot.com` and `http://forge.novell.com/modules/xfmod/project/?gcc-cil`, 2005.
17. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–335, 1997.
18. Tim Lindholm and Franck Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, April 1999.
19. Jason Merill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *GCC Developers Summit*, 2003.
20. Ricardo Fernández Pascual. HiPEAC PhD Internship: Ricardo Fernández Pascual – GCC CIL Frontend. `http://www.hipeac.net/node/823`, 2006.
21. Bernhard Rabe. Towards a CLI Assembly Format for Embedded Systems. In *International Conference on Embedded Systems and Applications*, Las Vegas, NE, USA, June 2006.

22. Stefan Richter, Andreas Rasche, and Andreas Polze. Hardware-near Programming in the Common Language Infrastructure. In *Proceedings of the 10th IEEE International Symposium on Object and Component-oriented Real-time Distributed Computing*, Santorini Island, Greece, May 2007. IEEE.
23. Jeremy Singer. GCC .NET - a Feasibility Study. In *1st International Workshop on C# and .NET Technologies on Algorithms, Computer Graphics, Visualization, Distributed and WEB Computing*, Plzen, Czech Republic, 2003.
24. STMicroelectronics. *STb7100-MBoard Datasheet – Single-chip, low-cost HD set-top box decoder for digital TV, digital set-top box or cable box*, December 2005.
25. Operating Systems and Middleware Group. Lego.NET website. `http://www.dcl.hpi.uni-potsdam.de/research/lego.NET`.
26. Martin v. Löwis and Jan Möller. A Microsoft .NET Front-End for GCC. In *Proceedings of .NET Technologies'2006*, pages 17–20, Plzen, Czech Republic, June 2006. ISBN 80-86943-11-9.