Comparing the Size of .NET Applications with Native Code

Roberto Costa
STMicroelectronics
Via Cantonale 16/E
6928 Manno, Switzerland
roberto.costa@st.com

Erven Rohou
STMicroelectronics
Via Cantonale 16/E
6928 Manno, Switzerland
erven.rohou@st.com

ABSTRACT

Byte-code based languages are slowly becoming adopted in embedded domains because of improved security and portability. Another potential reason for their adoption is the reputation for smaller code size than native. This is critical in contexts in which a small memory footprint is crucial to reduce production costs. This paper compares the code size of applications compiled for .NET framework with the same natively compiled for various processors. The paper shows that the assumption of an impressive code size reduction is not reachable and it suggests that the adoption of such languages in embedded contexts be justified by additional arguments. The paper also studies the reasons for this and it compares with the compression ratios achievable for various applications through alternative techniques.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization, Run-time environments*

General Terms

Measurement, Experimentation, Languages

Keywords

Bytecode, code size, .NET, managed environments

1. INTRODUCTION

Microsoft .NET is a framework that defines a platform independent format for executables and a run-time environment for the execution of applications. Parts of the framework have then been standardized by the European Computer Manufacturers Association (ECMA) and by the International Organization for Standardization (ISO).

The core of the framework is the *Common Language Runtime* (*CLR*), a runtime engine that provides any .NET application with services like machine independence, cross-language interoperability, code access security, memory management and garbage collection, thread facilities.

.NET executables are encoded in a *Common Intermediate Language* (*CIL*, sometimes called *MSIL* by Microsoft), an instruction set both language and machine-independent. The independence from the programming language is achieved by defining a *Common Type System* (*CTS*), a system of types adopted by all .NET-compliant programming languages, and a full set of standard libraries available at run-time. These libraries offer a wide set of functionalities, covering basic data structures, network services, database connectivity, graphical interface, localization, reflection and thread support.

The independence from the machine is guaranteed by the fact that CIL is not bound to the instruction set of the machine on which .NET applications are executed. Since a .NET application does not contain native code, it is not directly executable. One of the most important services made available by the CLR is the ability to translate CIL code into native code just before its execution. This kind of translation is called *JIT* (just-in-time). Though not specifically designed for it, CIL code can also be interpreted; however, this can significantly reduces the execution performance and is not desirable on code executed often.

Tools compliant with .NET framework produce code specified in CIL, that uses the CTS and that relies upon the standard libraries.

.NET has many similarities with Java [8, 13], a framework proposed by Sun in 1995. Java achieves the same purposes of platform-independence and security of applications through identical concepts. As a matter of fact, Java applications are specified in Java byte-code, a machine-independent instruction set executed by Java virtual machine, and rely on the classpath, a standard set of libraries.

Put aside these similarities, .NET has a few key differentiating features. First, unlike Java, .NET is designed to support several programming languages and to offer full interoperability among them. Second, .NET framework is an open standard, published by ECMA [7] and by ISO [10]. Third, .NET code is designed to be effectively translated just-in-time as opposed to be interpreted.

Byte-code based frameworks initially built a reputation of being slowly and inefficiently executed and of not being adapted to real-time contexts because of the overhead of the virtual machine services (i.e.: garbage collection). This prevented them at first from being used in the embedded world.

This reputation is no longer deserved. Since then, just-in-time compilation techniques have much evolved and in-

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

cremental algorithms for garbage collection improved [22]; in addition, languages like C do not even need any garbage collection. Sun supplies several versions of Java framework especially tailored for embedded domains [11] and Java applications are successfully being used in digital assistants, mobile phones, set-top-boxes and SIM-based smart cards.

.NET is attractive in the embedded domain for security and portability. The possibility of targeting C language (by eventually sacrificing some services) is another big advantage that facilitates the support of legacy code.

In addition to these advantages, a claim commonly accepted is that byte-code instruction sets lead to significantly smaller code than native binaries. In the embedded domain, code size is much more critical than for desktops because of the high cost of flash and ROM memories and of lack of big mass storage devices. Code reduction may represent yet another key reason why in the next generation embedded devices byte-code and dynamic translation techniques are going to play a major role.

The goal of this paper is to verify up to which degree the myth of code size reduction brought by byte-code holds.

Empirical results for a meaningful set of multimedia and general-purpose applications are shown and explained for CIL; x86 and ARM Thumb codes are also considered as other terms of comparisons. Finally, code compression is taken into account as an orthogonal way to reduce the code size, in order to check if CIL and native code behave differently in this respect.

The results show that, while CIL instruction streams are in average from 50% to 70% smaller than native code, they also carry (and depend on) higher-level information that occupies storage space (the metadata section); by counting it, the overall size of an algorithm implementation in CIL and in native instruction set is comparable.

Despite order-of-magnitude code size savings are out of reach and left to urban legends, the value of .NET framework in the embedded world is fully confirmed by the experimental findings. With a comparable code size, CIL code includes the additional information required by services more and more crucial (i.e.: security) without leading to a code size expansion.

Section 2 reviews related work. In Section 3 we present the most important features of a .NET executable. Our experimental setup is detailed in Section 4 and we present our results in 5. We conclude in Section 6.

2. RELATED WORK

Despite the youth of the framework, there is a large body of literature about .NET. ECMA specification [7] is the place to start from; then, many books have already been published [3, 19].

Following the existence of an open standard, .NET has risen an interest in the open source community. For instance, two open-source in-progress implementations of .NET framework are Mono [15] and Portable.NET [6].

It is a common belief that, in general, byte-code is more compact than native code for most machines. Nonetheless, to the best of our knowledge, there is no work that supports this conjecture or that publishes quantitative results about it.

Given the importance of memory footprint reduction for embedded applications, several code compression techniques have recently appeared. In order to increase the code density, modern ARM processors understand Thumb [20] as well, a separate 16-bit instruction set; MIPS' solution is very similar with the introduction of MIPS16e instruction set [14]. Additional reduced instruction sets usually have a price in terms of performance and they allow achieving a code size reduction of about 20 - 30%.

PowerPC architecture introduced CodePack [17], an on-the-fly decompression system. The compression scheme separates the opcode stream from the other bits of the instruction encoding. On average, it saves 20 - 30% of code size. The SlimCode [18] compression technique for ST200 directly compares with CodePack and it achieves an average compression ratio of 40%.

[12] compresses Java byte-code by using canonical Huffman codes; no decompression is needed. A 40% compression ratio is achieved with minimum slowdown.

3. OVERVIEW OF .NET APPLICATIONS

From a high level point of view, a .NET application consists of two parts: the actual code, or CIL, and the metadata.

CIL is a stream a byte-codes similar to a processor instruction set. Most of the opcodes are 1-byte long, a few are 2-byte long. One major difference with a processor instruction-set architecture (ISA) is that the evaluation of expressions uses a stack. A function consists of a header, a body and possible a footer. The header is 1 or 12 bytes. When present, the footer describes how exceptions should be handled. The function body contains the stream of byte-codes.

Metadata is a structured way to encode all the bits and pieces needed by the run-time engine to properly execute the application, namely to enforce security, dynamically resolve symbols, lay out classes, etc. Metadata contains the description of the classes and structures contained in .NET executable, the method and field information of classes, the dependencies on other modules, and much else. These sections do not concern only applications compiled from object-oriented languages like C# or C++; C programs compiled for .NET framework generate a good amount of such information as well.

Since CIL is a stack-based language, it does not need to encode registers. A typical arithmetic operation fits on 1 byte, whereas a typical 32-bit processor needs 4 bytes. However, the usage of an evaluation stack imposes constraints on the code, and more instructions might be needed to achieve the same computation. Section 5 gives more details.

It is important to understand that the metadata is not optional information added for the convenience of other tools (like debug information for example). It is necessary for the proper execution of the code. In other words, it has to be put in the flash memory of an embedded system.

4. EXPERIMENTAL SETUP

The experiments are performed on a set of C-language benchmarks composed of a mix of dedicated and general purpose applications, as reported in Table 1. aes4 and igp are proprietary implementations of image processing or encryption algorithms respectively. vvideo is a large video encoding proprietary application. Many are classical and publicly available algorithms.

This paper is interested in the evaluation of the code size for embedded systems, with respect to native code. The proprietary algorithms are optimized for embedded systems.

The benchmarks are compiled for .NET framework, for Intel x86 architecture and for Thumb [20], the 16-bit compact instruction set available in modern ARM processors. Compiler flags that optimize for code size are specified, when available; in particular, loop unrolling, software pipelining and inlining are disabled. Microsoft Visual C++ Toolkit 2003 and Microsoft Visual Studio 8 Beta are used as reference tools for .NET compilation; compilation flag `/O1` (minimize space) is selected. Other compilers used are gcc 3 for x86 and ARM ADS 1.2 for Thumb (with `-Ospace` command-line option, which reduces the image size at the expense of performance).

source	name	description
ST	aes4	aes encryption
	igp	graphic quality improvement
	adpcmc	adpcm audio encoder
	adpcmd	adpcm audio decoder
	anagram	anagram finder
	cjpeg	jpeg still picture encoding
	compress	standard UNIX utility
	dijkstra	Dijkstra's shortest path alg.
	djpeg	jpeg still picture decoding
	fp	integer floating-point emul.
	ks	graph partitioning algorithm
	wc	standard UNIX utility
	yacr2	routing algorithm
idSoftware	quake2	Quake2 in .NET
	vvideo	Video encoding

Table 1: Benchmarks

5. RESULTS

This section presents the code size obtained for the various considered architectures (subsection 5.1), followed by the reasons that explain the differences (subsections 5.2 and 5.3). Since the metadata is a significant chunk of the total .NET size, subsection 5.4 gives insights about its contents. Finally, the code size gains brought by .NET are compared with reasonable code compression ratios (subsection 5.5).

5.1 Raw Numbers

Table 2 shows the code size of the benchmarks compiled for the various architectures considered. The code size is computed as the sum of the sizes of the functions in the executable that correspond to C functions in the source file (i.e.: run-time support static libraries linked within the executable are not counted). Particular care is used to make sure that no function is inlined. For .NET, the sizes of CIL and metadata are separately shown; padding bytes present in .NET executables are not counted.

In spite of native instruction-set architectures' reputation of poor code density, native code compares favorably with .NET executables. While CIL size typically varies from 60 to 70% of native code size, the situation changes when metadata too is counted. This is especially true for small benchmarks, showing that a minimum size of metadata is about 2000 bytes; beyond this threshold, the metadata size grows more or less linearly as CIL.

x86 code is more compact than many embedded cores on general-purpose algorithms, in which it confirms its fame of very dense CISC instruction encoding. In multimedia code the situation is different and emdedded processors typically outperforms x86. The reason is that these algorithms are well suited for architectures with many registers and the very few visible registers of x86 ISA force it to spill many temporary variables to memory.

Finally, Thumb code proves to be the most compact by far: even without counting .NET metadata, Thumb executables are significantly smaller than CIL in all benchmarks.

	.NET				
bench.	CIL	meta	total	x86	Thumb
adpcmc	581	2184	2765	831	n/a
adpcmd	520	2188	2708	702	n/a
aes4	2092	3156	5248	2272	1224
anagram	2117	4684	6801	2845	n/a
cjpeg	115897	83452	199349	148124	n/a
compress	4661	6992	11653	6234	n/a
dijkstra	943	3256	4199	1217	558
djpeg	115849	84496	200345	148607	n/a
fp	5012	4176	9188	5653	n/a
igp	3047	2984	6031	5910	2204
ks	3991	7428	11419	5342	3440
wc	204	1796	2000	306	160
vvideo	152600	45788	198388	249707	n/a
yacr2	14399	11264	25663	24170	11800

Table 2: Code Size Comparison

5.2 Reasons that could make CIL larger

There are a number of reasons that can make CIL code larger than native code. We identified some that are inherent to the language specification. We also investigated the role of the compiler in the generated code.

5.2.1 Address Computation

Many instruction sets support several addressing modes for memory operations. For instance, the Intel x86 ISA has a very wide range, allowing a single address to be composed of a base, an offset and a scaling factor. On the other hand, .NET supports only one addressing mode for `ldind`: load the value whose address is on the evaluation stack. As a consequence, some addresses have to be explicitly computed in CIL.

5.2.2 Lack of Conditional Assignment

CIL lacks of any conditional assignment instruction or any form of predication. Many modern processors implement a limited form of if-conversion, which simplifies the control-flow graph when applied. CIL code must instead contain the code corresponding to the original control-flow.

For instance, let us consider the following C statement:

```
e = (a < b) ? c : d;
```

The corresponding CIL code takes up to 14 bytes, while conditional assignment could simplify the control-flow and reduce the code to 8 bytes for a typical 32-bit machine (see

Figure 1, the right side shows pseudo-code for a 32-bit processor with conditional assignment).

size	code
2	ldloc.s a
2	ldloc.s b
2	bge.s → L1
2	ldloc.s c
2	br.s → L2
	L1:
2	ldloc.s d
	L2:
2	stloc.s e

size	code
4	compare p = a, b
4	select e = p, c, d

Figure 1: Lack of conditional assignment

5.2.3 Conversions

CIL is a strongly-typed language, there are only a few implicit coercions and all the other type conversions are explicit. Consider the following piece of C code:

```
char x, *p;
*p = x + 1;
```

The C language standard [9] specifies that the addition be performed on `int` type. That is: `x` has to be converted to `int`, before 1 is added. The result is then converted back to `char`. CIL does pretty much the same. On the contrary, the native instruction set is not typed. Instead, x86 `movb` instruction is a *move-byte* instruction that stores only one byte to memory, making the conversion useless. This same example also shows the difference in terms of address computation between CIL and x86. CIL code takes 8 bytes, while 5 bytes are sufficient for x86.

size	code
2	ldloc.s p
2	ldloc.s x
1	ldc.i4.1
1	add
1	conv.i1
1	stind.i1
	CIL

size	code
3	add $0x1,%ebx
2	movb %ebx,0x1(%eax)
	x86

Figure 2: Type Conversions

5.2.4 Maturity of Compilers

Surprisingly, even a quick visual inspection of the generated byte-code reveals missed opportunities for optimizations. Consider the code of Figure 3. Some value is computed and stored in the local variable 2. Then, the value of this variable is retrieved and used. This is the only use of the variable. Clearly, the compiler could have optimized away the `stloc` and the `ldloc` and directly used the value on the stack. This optimization is similar to register promotion [16].

Another simple optimization regards the allocation of local variables. The variables with indices 0 to 3 can be accessed with one byte (`ldloc.0` - `ldloc.3`). Other variables require two (`ldloc.s`) or four bytes (`ldloc`). It is straightforward to allocate the most used values to the first four

size	code
..	*compute value*
1	stloc.2
1	ldloc.2
..	*last use of value*

Figure 3: Missed "register promotion"

variables. Portable.NET compiler currently fails to do it, which results in code larger than necessary; Microsoft compilers do not miss this optimization.

In some cases, the code generated by the compiler could be made much smaller, possibly at the cost of a slower execution. Consider the example of Figure 4: the code is loading the constant value 1.0 on the stack. The straightforward way to do this is to use a `ldc.r8` operation, with the value inlined on the following 8 bytes. A much smaller way to do it is to load the integer value 1 and convert it to float. One could argue that this might make the execution slightly slower if the JIT is not able to recognize the pattern, but the real impact is target dependent and it is debatable whether CIL should be written with a specific target in mind.

size	code
9	ldc.r8 1.000

size	code
1	ldc.i4.1
1	conv.r8

Figure 4: Loading a floating point value

Finally, in rare circumstances Microsoft Visual C++ Toolkit 2003 C compiler applies control-flow graph simplifications that increase the code size, like peeling part of a loop iteration. When code size reduction is the main concern, such transformations should be disabled.

5.3 Reasons that could make CIL smaller

There are also reasons that tend to make CIL smaller than traditional native code.

5.3.1 Evaluation Stack versus Registers

The typical CIL opcode is 1 byte long. CIL is stack-based: operands are not explicitly encoded in the instruction and are expected to be at the top of an evaluation stack. This makes it possible to use very short operations.

However, when operands are not in the right stack position, they have to be explicitly copied on top of it. Some code intrinsically requires more such copies because of its data-flow structure. Therefore, the code size reduction due to this reason much depends on the kind of code.

5.3.2 Floating Point

Many embdedded processors lack of floating point units, and all computations are simulated in software. On the other hand, CIL provides polymorphic operators, like `ADD`, `MUL`, etc., which apply on integer as well as on floating point operands. This means that CIL simply emits one byte for a floating point operation (assuming the operands are already on the evaluation stack). A core without a floating point unit has to emit a call to an intrinsic function, and to place the operands in the proper registers according to the ABI. This requires extra register transfers.

5.4 Metadata

The metadata contains symbolic information about the objects contained in the executable. It consists of four *heaps* and several *tables*. The heaps are named #Strings, #GUID, #US (for user string), and #Blob. See [7] for a complete specification of the metadata.

Table 3 shows the metadata size for the benchmarks and the contribution of each heap to the total. Since #US always empty or comparatively very small, it is omitted in the figure. Similarly, #GUID is empty or contains a 16 byte signature.

Almost half the size of a .NET executable is used by the #Strings heap. In brief, it contains the symbolic names of all the entities present in the module (type and class names, methods, fields, etc.) as defined by the programmer. [2] made a similar experiment on the Java byte-code and found a similar result: the size of the constant pool is 60% of the total size on average, and 60% of it is what they called *type and link information*, i.e. what is needed for the run-time type checking and dynamic linking.

benchmark	metadata	tables	Strings	Blob
adpcmc	2184	40%	41%	14%
adpcmd	2188	40%	41%	14%
aes4	3156	38%	46%	13%
anagram	4684	42%	42%	13%
cjpeg	83452	32%	55%	13%
compress	6992	40%	47%	12%
dijkstra	3256	41%	43%	12%
djpeg	84496	32%	55%	13%
fp	4176	36%	54%	7%
igp	2984	38%	39%	19%
ks	7428	35%	50%	13%
wc	1796	37%	43%	14%
yacr2	11264	34%	52%	13%
quake2	233036	37%	56%	7%

Table 3: Distribution of Size in Metadata

In many cases, the symbolic names are quite long, especially when they come from C++ mangled names [4]. Often, the names could be shortened to a few bytes, drastically reducing the heap size. Attention must be paid to those used by external modules for linking, and to those used by reflection [5].

Some commercial tools exist that strip the metadata to reduce the code size, for example Xenocode [1].

5.5 Compression

An obvious way to address code size is to use compression. Many techniques have been proposed, from pure hardware to pure software [14, 12, 17, 18, 20].

Compression techniques are closely related to information theory and entropy. [2] measure the entropy of a large set of Java byte-codes by using Shannon's formula:

$$H = -\sum_i p_i \times \log_2 p_i$$

They find it to be between 3.2 and 3.6. The same formula, applied to the benchmarks compiled for the .NET environment, yields values close to 5 (see column H on Table 4). Applied to the opcode stream of embedded applications, typical values for the formula are from 4 to 6.

In order to evaluate how much space can be saved with compression, the experiment consists in extracting the code from the executable, and compress it with the standard tool `bzip2` [21], at maximum compression level. Table 4 shows the results. Please, notice that x86 datas refer to the overall size of the text section of the final executable, which typically includes statically linked code that is not part of the application; this explains why the figures slightly differ from those published in Table 2. The compression ratios for x86 are not significantly affected by that.

This approach works reasonably well for .NET executables, which are composed of a stream of bytes. Higher compression rates could be achieved for native executables by taking into account the instruction encoding, like done in [17, 18]; the reason is that opcodes and registers have different and often uncorrelated dynamics. Instruction-encoding sensitive compression for STMicroelectronics ST200 is exploited in [18], with improved compression ratios.

6. CONCLUSION

This paper studies the code size of applications developed in the .NET framework. It finds that the metadata is a substantial part of the total size. Several aspects of the CIL representation that differ from traditional native instruction sets and that have an impact on size are analyzed. The findings show characteristics similar to a previous study on Java [2].

The code that is needed to run an application is not only the CIL, but also the metadata. Contrarily to a common belief, the total code size is not significantly more compact that a native ISA, and it is actually much larger than ARM/Thumb.

Still, areas not yet covered by the existing tools are identified, which should further reduce the code size; they might be promising directions when using .NET in embedded systems.

7. REFERENCES

[1] XenoCode Corp. `http://www.xenocode.com`.

[2] D. N. Antonioli and M. Pilz. Analysis of the java class file format. Technical Report 98.4, Department of Computer Science, University of Zürich, April 1998.

[3] D. Box and C. Sells. *Essential .NET Volume 1: The Common Language Runtime*. Addison-Wesley, 2003.

[4] CodeSourcery, Compaq, EDG, HP, IBM, Intel, R. Hat, and SGI. Itanium C++ ABI. `http://www.codesourcery.com/cxx-abi/abi.html`, 2001.

[5] D. Curran, F. C. Ferracchiati, S. F. Gilani, M. Gillespie, S. Gopikrishna, J. Hart, B. K. Mathew, A. Olsen, J. Pinnock, T. Titus, and S. Sivakumar. *Pro .NET 1.1 Remoting, Reflection, and Threading*. Apress, 2005.

[6] DotGNU Portable.NET. `http://www.dotgnu.org/pnet.html`.

[7] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*, 2nd edition, Dec. 2002.

[8] Gosling, Joy, Steele, and Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, Apr. 1999.

[9] International Org. for Standardization. *ISO/IEC 9899:1990: Programming languages — C*. 1990.

benchmark	x86			.NET			
	orig	bzip2	ratio	orig	bzip2	ratio	H
adpcmc	1201	760	37%	581	462	20%	5.13
adpcmd	1171	741	37%	520	432	17%	5.16
aes4	3851	1435	63%	2092	819	61%	4.70
anagram	2872	1504	48%	2117	1269	40%	5.18
cjpeg	149138	47470	68%	115897	44335	62%	4.96
compress	6297	2656	58%	4661	2369	49%	5.30
dijkstra	1217	677	44%	943	517	45%	4.90
djpeg	149618	47594	68%	115849	44470	62%	4.97
fp	12147	3467	71%	5012	1748	65%	4.37
igp	5910	1951	67%	3047	1381	55%	4.43
ks	5342	1987	63%	3991	1852	54%	4.83
wc	306	278	9%	204	214	-5%	4.54
vvideo	263669	66977	75%	152600	60129	61%	5.04
yacr2	24170	5917	76%	14399	5907	59%	4.99

Table 4: Compression ratios of .NET and native code

[10] International Org. for Standardization. *ISO/IEC 23271:2003: Common Language Infrastructure*. 2003.

[11] Java 2 platform, micro edition (J2ME). `http://java.sun.com/j2me`.

[12] M. Latendresse and M. Feeley. Generation of fast interpreters for Huffman compressed bytecode. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 32–40, 2003.

[13] Lindholm and Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, June 2000.

[14] *MIPS32 Architecture For Programmers, Volume IV-a: The MIPS16e Application-Specific Extension to the MIPS32 Architecture*. MIPS Technologies Documentation, 2001.

[15] Mono. `http://www.mono-project.com`.

[16] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[17] A. Orpaz and S. Weiss. A study of CodePack: optimizing embedded code space. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 103–108. ACM Press, 2002.

[18] E. Piccinelli and R. Sannino. Code compression for VLIW embedded processors. *ST Journal of Research*, 1(2):32–46, 2004.

[19] J. Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.

[20] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley, 2nd edition, Dec. 2000.

[21] J. Seward. bzip2. `http://sources.redhat.com/bzip2`. bzip2 is a freely available, patent free, high-quality data compressor.

[22] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.