Common Language Runtime: a new virtual machine

João Augusto Martins Ferreira

Departamento de Informática – Universidade do Minho
4710-057, Portugal
joao.ferreira@progmat.com

Abstract. Virtual Machines provide a runtime execution platform combining bytecode portability with a performance close to native code. An overview of current approaches precedes an insight into Microsoft CLR (Common Language Runtime), comparing it to Sun JVM (Java Virtual Machine) and to a native execution environment (IA 32). A reference is also made to CLR in a Unix platform and to techniques on how CLR improves code execution.

1 Introduction

Virtual machines, intermediate languages and language independent execution platforms are not a recent technology and we heard talking about it . Backing in time we see UNCOL[1], USCD P-code[2], ANDF[3], AS-400, some hardware emulators like VMWare or Transmeta Crusoe, binary translation [2], the JVM [3] and more recently the Microsoft implementation of Common Language Infrastructure (CLI) [4].

The word **"virtual"** is used because, generally, it is a technology that is implemented in software on top of a "real" hardware platform and operating system. All virtual machines have a programming language (or more) that is compiled and will only run on that virtual platform. The virtual machine plays a central role because it provides a layer of abstraction between a compiled program and the underlying hardware platform and operating system. So, this is great for portability because any program that targets the virtual machine will run, whatever maybe underneath a particular virtual machine implementation.

This is great comparing to native compilers and execution. For instance, runtime code generation for C must take into account the machine architecture, operating system conventions and compiler-specific issues such as calling conventions [5]. Trying to avoid this platform dependence, the goal was to develop a language that will run as a bytecode execution, but the generated code will not run at native speed[4]. However, the new virtual machines, such as Sun's Java Virtual Machine and Microsoft's CLR, use JIT (Just In Time) compilers that generates well-optimized native machine code from the

---

[1] Universal Computer Oriented Language was a universal intermediate language, discussed but never implemented. [1]

[2] P-Code was a VM of historical note as an early OS for some microcomputers. It was based on Pascal's operational model and implemented in UCSD Pascal witch was also the most used language on the platform. For further information see http://www.threedee.com/jcm/psystem/

[3] The ANDF (Architecture Neutral Distribution Format) technology for portable binary code distribution is a VM developed by the OSF and now TrenDRA (www.trenda.org) project is derived from it. It's not based on a low-level virtual stack or register processor but rather on some kind of abstract syntax free for a simple low-level algebraic language. This portable intermediate code then gets compiled by an architecture specific program on the host that runs the code.

[4] This was the main reason for the failure of the most Virtual Machines.

bytecode at runtime. These platforms combine the advantages of bytecode generation (portability) with native code generation (speed) and they also include a bytecode verifier that detects type errors and other flaws before executing the bytecode.

Like we said, a programming language (high level) is created to run in a specific virtual machine. For instance, Java was created to target Java Virtual Machine and then gets some advantages about its processing. However, CLR will use CLI (Common Language Infrastructure) that was designed from the scratch as a target for multiple languages.

This paper shows how CLR (is nothing more than a virtual machine) executes and uses CLI to generate the code to run on a native machine. It also analyse some of its advantages, namely portability, compactness, efficiency, security, interoperability, flexibility and, above all, multi-language support.

We will try also to show CLR implementations on non-Windows operating systems namely the Mono Project (Unix) and DotGnu, together with some techniques that CLR uses to improve code execution.

2 CLI and CLR architectures

The CLI is an International Standard [2] that is the basis for creating execution and development environments in which languages and libraries work together seamlessly. The CLI **specifies** a **virtual execution system** that shields CLI-compliant programs from the underlying operating system. Comparing to other virtual execution systems that were developed for different operating systems, programs written with CLI-compliant languages can be run in these different systems without recompiling or even rewriting.

The CLR is a real implementation of CLI that is owned and controlled exclusively by Microsoft. So, we can say that CLR is a virtual machine that will translate CLI-compliant programs to native code, like every virtual machine do.

The architecture of CLI is divided on the following elements:

- Common Type System (CTS)
- Common Language Specification (CLS)
- Common Intermediate Language (CIL) instruction set
- Virtual Execution System (VES) with executes managed code[5] and lies between code and the native operating system

In one hand, CTS defines the complete set of types available to a CLI-compliant program. On other hand, CLS defines the subset of CTS types that can be used for external calls[6]. Any language will communicate with each other through the VES using **metadata,** nothing more that the information with the code and data that describes the data, identifies the locations of reference to objects and gives information to the VES, that will handle all the major overheads of traditional programming models (exceptions, security concerns, performance, pointers, object lifecycle, etc.). CIL is an assembly like language that is generated by compilers of languages targeting CLI. How and when the CIL is compiled to machine code are not specified by the standard and those determinations rest with the implementation of the VES. The most frequently used model is just-in-time (JIT) compilers that generate native code as it is needed. Install time compilers are another option and it is also possible to implement an interpreter, rather than a compiler, for the CIL.

[5] Managed code is the code CLI compliant and unmanaged code is the non CLI-compliant code.
[6] This is done because each programming language that compiles with the CLI uses a subset of the CTS that is appropriate for that language.

All virtual machines have registers, but they are mostly **stack based**: all the operands and instructions are directly handled in the stack [6]. This stack approach is better than registered based since work is done by the runtime machine.

The success of CLR is based on how Microsoft created a VES for CLI. Also, it has created languages to target CLR (C#, VB.NET, C++.NET) and built a Base Class Library that programmers can use to, easily, perform their normal tasks and it is available, also, to all the languages that are CLI-compliant. Also, any company can design a compiler to produce CIL code that will be interpreted or compiled by the CLR.

The developing of an application, traditionally, starts with the writing of the source code and then compiling and linking the source code into an executable. CLR introduces the concept of **managed module**. A CLR module is a byte stream, typically stored as a file in the local file system or a Web Server [8].

Basically, CLR modules contain: A PE Header (this standard Windows 32 bit (not 64!) Portable Executable (PE) file header, which is similar to Header the Common Object File Format (COFF) header; A CLR Header (contains the information (interpreted by the CLR and utilities) that makes this a managed module); Metadata (every managed module contains metadata tables. There are two main types of tables: tables that describe the types and members defined in your source code (DefTables) and tables that describe the types and members referenced by your source code (RefTables); Intermediate Language code (code that the compiler produced as it compiled the source code. The CLR later compiles the IL into native CPU instructions); resources (consist of static read-only data such as strings, bitmaps and other aspects of the program that are not stored as executable code).

So, modules are not more than physical constructs that exist as byte streams, typically in the file system.

An **assembly** is a configured set of loadable code modules that together implement a unit functionality [2]**.** Informally, an assembly is an implementation of the component concept[7]. Normally, an assembly figures on operating system as an executable file (.EXE) or an library file (.DLL). The assembly **manifest** describes information about the assembly itself, such as its version, which files make up the assembly, which types are exported from the assembly and can have also a digital signature and a public key of the manifest [7].

Certainly, you are asking about how an assembly, containing managed code, can execute on a non-management environment like the Windows 32 API. The answer lies in the flexibility of the PE files.

To show this we will write a simple program and then compile it[8]. Here the source code:

```
using System;
class Hello
{
    public static void Main()
    {
        System.Console.WriteLine("Hello world!");
    }
}
```

After compiling it (csc hellocworld.cs), for analysing the PE, we will use PEDump[9]. When the assembly is executed (helloworld.exe) the first task performed is to know what

[7] Any piece of software or hardware that has a clear role and can be isolated allowing you to replace it with a different component with equivalent functionality.

[8] Note that when compiling the source code with an high level language, the compiler will produce IL code that is turned into instructions that are specific to the machine on which the IL code is loaded by the Just-In-Time (JIT) compiler.

additional modules are required for this executable to run. The import address will tell us that:

```
IMPORT        rva: 00002290  size: 0000004B
```

An RVA (Relative Virtual Address) points to an area in one of the sections of the file. To decode where an RVA points, you first find in which section the RVA is (.text, .rsrc, .reloc, and so forth). Each section has a start address (virtual address) and a size. If the RVA is greater than the start address and less than the start address plus the size, then the RVA is pointing to an address in that section. Subtract the start address of the section from the RVA, and that forms an offset into the section. This import address points to load the mscoree.DLL, the **Microsoft .NET Execution Engine**. Since then, all managed modules related to this component will be loaded. This causes execution to start in the execution engine. After the execution engine loads the CLR (the workstation version `mscorwks.dll`), the assembly manager (`fusion.dll`), the CLR class library (`mscorlib.dll`), the strong name support (`mscorsn.dll`), and the JIT compiler (`mscorjit.dll`), the assembly that is being run is queried for where managed execution should begin. To know where the execution should begin the CLR will look in the **CIL Header** for an entry called Entry Point Token[10].

```
COM_DESCRPTR      rva: 00002008  size: 00000048
```

Then, the loader will inspect for the method table that refers to the C# entry Main. As part of the method table, one entry refers to the address of the IL code that makes up this method. The address points to the following bytes in the assembly:

```
000050:  13 30 01 00 0B 00 00 00  00 00 00 00 72 01 00 00 .0..........r...
000060:  70 28 02 00 00 0A 2A 00  13 30 01 00 07 00 00 00 p(....*..0......
```

After setting up the header for the method, these bytes translate into the following IL instructions:

```
ldstr 0x70000001
call 0xA0000002
ret
```

The `ldstr` instruction loads a string token (0x7 indicates the user string table, and index 1 of the table refers to the string "Hello World".) The call instruction makes a call into a referenced method (0xA refers to the member ref table, and index 2 indicates it is the `WriteLine` method). The last instruction is a return that finishes this method. The JIT compiles these instructions into native code, at which point they are executed and the program terminates.

3 IL: Instruction Set and JIT

The CLI has about 220 instructions so it's impossible to cover all in this paper[11]. Comparing to JVM [7] CLI instructions are more polymorphic and only requires explicit

---

[9] See the article on how to get headers from Win 32 executables, in http://msdn.microsoft.com/msdnmag/issues/02/02/pe/default.aspx

[10] CIL Header is known at the PEDump program as COM_DESRPTR.

[11] To see the ISA of CIL see the ECMA standard – Partition III.

type information for the result of an instruction. Although, it requires more work from the JIT. Instructions that push values on the evaluation stack are called “loads” and instructions that pop elements into local variables are called stores. The simplest load instruction is *ldc.t v*, that pushes the value *v* of type *t* on the evaluation stack. The *ldnull* pushes a null reference on the stack. The *ldarg n* instruction pushes the contents of the n-th argument on the evaluation stack. The *ldarga n* instruction pushes the address of the n argument on the evaluation stack. The *ldarg.1* is used to reference arguments. Local variables are referenced by *ldoc.0* (zero-based) The *stloc n* instruction pops a value from the stack and stores it in the n-th argument.

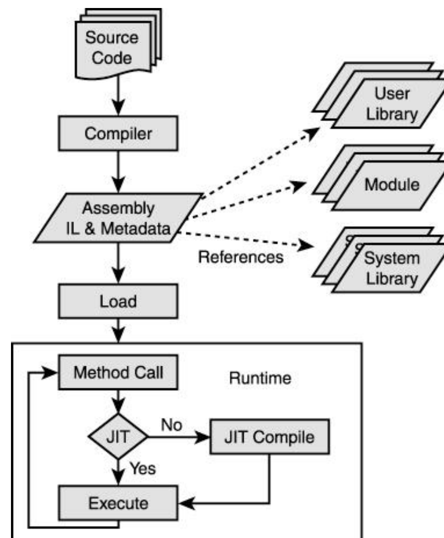

Figure 1 – Runtime execution of an assembly.

To see some real IL code we can compare it with simple C# instructions and how the are compiled to IL code.

For example the following IL code is used to push an argument (argument 1) onto the stack and store it in a variable (location zero):

```
IL_0000:  ldarg.1
IL_0001:  stloc.0
```

One more example, shows the IL code to add two numbers:

```
IL_0002:  ldloc.0
IL_0003:  ldarg.2
IL_0004:  add
IL_0005:  stloc.1
```

This loads what is at location 0 (perhaps a variable), loads argument number two of the function signature and then adds them. Finally this result is stored at location 1 (maybe another variable).

And how IL will fit on CLR execution?

Well, observing figure 1 we see that source code is compiled into an Assembly (packing modules) containing IL and metadata (that references system library). When an assembly is loaded the not all the IL code will be turned into machine instructions. So, only the methods that are used will be turned machine code. If the JIT compiler has already compiled the code, then the code is executed. IF the JIT compiler has not yet seen the code, then the code is compiled and then executed. This process goes on until either all the IL code has been compiled or the application ends.

Because IL is "**intermediate**" it **cannot depend on a CPU execution model**. It cannot assume the registers of the CPU because IL doesn't know even the concept of registers. IL is completely stack based and, more over, its an OOP assembly language. Another advantage is that all the instructions are agnostic relating to the types (typeless), that is, it can load any type conforming to CTS. Unlike the Win32 loader, the CLR loader does not resolve and automatically load the subordinate modules (or assemblies). Rather, the subordinate pieces are loaded on demand only if they are actually needed (as with Visual C++ 6.0's delay-load feature). This not only speeds up program initialization time but also reduces the amount of resources consumed by a running program [8].

So, what the JIT will do?

1 Analyse the metadata. In the assembly that implements the type (Console), lookup the method (WriteLine) being called in the metadata;
2 From the metadata, get the IL for this method;
3 Allocate a block of memory;
4 Compile the IL code into native code; save the code in the memory allocated in step 3;
5 Modify the method's entry in the Type table so that it can now points to the memory block allocated in step 3;
6 Jump to the native code contained inside the memory block

4 JIT AND CLR performance

.NET provides two JIT compilers and a install-time JIT option: a "normal" JIT compiler, designed to provide the excepted optimization when compiling IL to native code; and ngen.exe, where MSIL code is "preJITed" and the native images are cached for future use.

Talking about performance we can always talk about pretty code techniques to improve execution. However, we will analyse some techniques on how JIT compilers improve execution over native platforms and how to improve the JIT execution.

As we already seen, the CLR needs a way to compile the intermediate language down to native code. When you compile a program to run in the CLR, your compiler takes your source from a high-level language down to a combination of MSIL (what Microsoft calls to IL language) and metadata. These are merged into a PE file, which can then be executed on any CLR-capable machine. When you run this executable, the JIT starts compiling the IL down to native code, and executing that code on the real machine. This is done on a per-method basis, so the delay for JITing is only as long as needed for the code you want to run.

The main thing about JIT is that it can analyse the code at runtime and generate better instruction of that [10]:

- Calculate constant values at compile time

Before	After
`i = 5 + 7`	`i = 12`

- Substitute backwards to free variables earlier

Before	After
`i = a`	`i = a`
`y = i`	`Y = a`
`z = 3 + y`	`z = 3 + a`

- Replace arguments with values passed at call time and eliminate the call

Before	After

(…)
x=foo(4, true);
(…)
}
foo(int a, bool b){
if(b){
return a + 5;
} else {
return 2a + bar();
}
	(…)
	x = 9
	(…)
	}
	foo(int a, bool b){
	if(b){
	return a + 5;
	} else {
	return 2a + bar();
	}

- Remove code inside loops if it is duplicated outside

Before	After

for(i=0; i< a.length;i++){
if(i < a.length()){
a[i] = null
} else {
raise IndexOutOfBounds;
}
}
	for(int i=0; i<a.length; i++){
	a[i] = null;
	}

- The overhead of incrementing counters and performing the test:

Before	After

for(i=0; i< 3; i++){
print("flaming monkeys!");
}
	print("flaming monkeys!");
	print("flaming monkeys!");
	print("flaming monkeys!");

- If a variable still contains the information being re-calculated, use it instead:

Before	After
x = 4 + y z = 4 + y	x = 4 + y z = x

JIT also can take out dead-code and take advantages on en-registration. Above all, JIT can perform some optimizations that a regular compiler can't, such a CPU-specific optimizations and cache tuning. The main advantage of JIT against a traditional compiler is

that JIT is activated at run time. This allows performing several optimizations that can be only done when the program is executed [10]. Using JIT, for example, can have processor specific optimizations since JIT knows when to use SSE or 3DNow instructions. The main thing here is that the code is compiled specially to P4 or Athlon, taking out the benefits on the ISA of that processor. JIT can also look at functions that are called several times since it is aware of control flow during run time.

However, these run time improvements come with a small gap, a one time start-up cost. The first time an IL application is executed is slow. After, the application will be faster, because the reasons we already explained. The only way to avoid this is using the tactic of precompiled code. CLR has a tool for that: the ngen.exe[12]. However, since the runtime optimization only occur when the programs executes, this approach is not so good like the code generated by a normal JIT. But, it will be better at the start-up time. When NGEN.EXE and MSCORPE.DLL generate a native image, it is stored on disk in a machine-wide code cache so that the loader can find it. When the loader tries to load a CIL-based version of an assembly, it also looks in the cache for the corresponding native image and will use the native machine code if possible. If no suitable native image is found, the CLR will use the CIL-based version that it initially loaded.

The use of precompiled code depends of the nature and the function ability of your application. You should use precompiled code if your application loads a lot of functions at start-up time. Also it can be fit to you if you are coding shared libraries, since you pay the cost of loading these much more often[13].

5 Unix with .NET

The Mono project is a community initiative to develop an open source, Linux based version of the Microsoft.NET development platform[14]. The main goal of this project is to add the C# language specified at ECMA[15] to other open source development tools and allow the creation of operating system independent .NET programs. This project is maintained by the Ximian (http://www.ximian.com) that also took part of the Gnome Project, a free Desktop for Linux. The main thing about the Mono Project is to show that is possible to create an universal platform that can run IL code everywhere and anywhere. They will try to create a C# compiler for Linux developers to make .NET compatible applications along with a complete implementation of class libraries compatible with the CLI. To run these applications, will be also necessary a Linux version of CLR and JIT (Just in Time Compiler), allowing the end-user to run .net applications built on Windows, Linux or other Unix Platforms. Now Mono ships a JIT compiler for x86 based systems. It is tested regularly on Linux, FreeBSD and Windows (NT core). There is also an interpreter, which is slower, that runs on the s390, SPARC and PowerPC architectures.

Another initiative is DotGnu[16] project. This is an effort to build an alternative to the .NET strategy of Microsoft. This project consists, actually, on three other projects: the DotGNU Portable.NET, the phpGroupWare and the DGEE. The DotGNU Portable.NET is completely compatible with ECMA 334, 355 specifications for C# and CLI respectly, and, the Microsoft CIL implementation. There is also a runtime engine (ilrun) which interprets programs with CIL bytecode format. However, instead of interpret CIL bytecode directly

---

[12] You can see NGEN.EXE as a deployment tool. CLR also has MSCORPE.DLL, an underlying library that generates with *ngen* native images at deployment time.
[13] For example, Microsoft pre-compiles CLR, because it is called several times.
[14] www.go-mono.com
[15] http://www.ecma.ch/ecma1/STAND/ecma-334.htm
[16] http://www.gnu.org/projects/dotgnu/

they convert, first the CIL into a simpler instruction set that they call (CVM - Converted Virtual Machine). The Portable.NET also has a compiler (`cscc`) that supports C# and C and it's designed to support bytecode generation for multiple bytecode systems (CLI, JVM e Perl 6 Parrot).

7 Conclusions

This communication presented various aspects about virtual machines and a solution provided by Microsoft with its CLR that is basically a new virtual machine running components, called managed assemblies.

We saw that is possible to design a virtual machine that can be implemented in a wide range of software and hardware platforms. The advantage of CLR over JVM lies on how it can be a target of a wide range of programming languages.

Also it is possible to achieve a good performance by using JIT compilation and that makes a platform independent code execution, since the virtual machine will transform an intermediate language to native code.

Finally, if you have a device or a hardware layer and you want portability, why not make a compiler targeting CLI and maybe design a VES that could run any program that is CLI-compliant on that device?

References

[1] Steel, T.B: A first version of UNCOL. Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference. 1961. 371 – 377

[2] Sites, R., Chernoff, A., Marks, M.: Binary Translation. CACM, 36(2):69–81, 1993.

[3] Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison Wesley, 1999.

[4] Microsoft: ECMA and ISO/IEC C# and Common Language Infrastructure Standards. 2003. http://msdn.microsoft.com/net/ecma/.

[5] Sestoft, Peter: Runtime Code Generation with JVM and CLR. Department of Mathematics and Physics, IT University of Copenhagen, 2002 (DRAFT)

[6] Simões, Alberto Manuel Brandão: Silicion Virtual Machines. Proceedings ICCA'03, Departamento de Informática, Universidade do Minho, 2003

[7] Meijer Erik, Gough, John: Technical Overview of the Common Language Runtime. Microsoft Research, 2002. http://research.microsoft.com/~emeijer/

[8] Box, Don, Sells, Chris: Essential .NET, Volume 1: The Common Language Runtime. Addison Wesley, 2002

[9] Burton, Kevin: NET Common Language Runtime Unleashed. Sams Publishing, 2002

[10] Schanzer, Emmanuel: Performance Considerations for Run-Time Technologies in the .NET Framework. Microsoft Corporation, 2001. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetperftechs.asp