Self-contained CLI Assemblies

Bernhard Rabe
Haso-Plattner-Institute,
University of Potsdam
P.O. Box 90 04 60
14440 Potsdam, Germany

bernhard.rabe@hpi.uni-potsdam.de

ABSTRACT

High-level programming languages and bytecode-based virtual execution environments have become popular in software development. Bytecode-based runtimes extend embedded system by techniques to improve safety, help portability and interoperability. The ECMA/ISO Common Language Infrastructure (CLI) specifies a bytecode-based execution environment (Common Language Runtime) and a comprehensive class library. CLI applications suffer from long startup time, high memory consumption and the amount of referenced assemblies. Startup time is determined by resolving references and high memory consumption through big class library assemblies. Often CLI applications use a small subset of the CLI class library, but the whole memory footprint is basically determined by the class library. To overcome memory requirements of the class library, a minimal application format that includes all essential class library functionality is reasonable. Self-contained CLI assemblies as an approach for size-optimized deployment format are presented in this paper.

Keywords

CLI, assembly format, space-optimization.

1. INTRODUCTION

High-level programming languages and bytecode-based execution environment have become popular in development of desktop systems. The *Common Language Infrastructure* (CLI) [Int03a] as implemented in the .NET Framework [Mic05a] has been a popular platform for creating component-based applications, because of:

- Platform independence of bytecode-based executables
- Fine granular security restrictions
- Revisable code
- Component model

It would be beneficial if CLI applications could be executed on memory restricted systems that are not covered by existing CLI implementation. .NET developers could then reuse their code for these systems instead of reimplementing their applications from the ground up using C or C++.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

Embedded systems differ from desktop systems in various aspects:

- Hardware resources are often limited: memory size, processing power, power supply.
- Software capabilities: Faulty programs can crash the system, because memory protection is not available.
- Capabilities for developer interaction, for debugging, or communication bandwidth are often limited.

CLI technology is integrated seamlessly in Rapid Application Development tools as Microsoft's Visual Studio suite for desktop development just as for embedded development. Compiler and tools are available for multiple programming languages e.g. C#, C++ .NET, or Delphi. The CLI could offer developers of embedded systems the same advantages as for desktop systems.

Due to the predictable nature of the sandbox-mode execution of CLI instructions, programming errors never result in system crashes, but cause exceptions

to be thrown. This allows for a simpler postmortem analysis of a fault. Due to the support for rapid prototyping, simulators for the target can be more easily created. Ideally, much of the code would only use standard library functions of the CLI, so that simulators are only necessary for the target-specific hardware.

The CLI as implemented in the Microsoft .NET Framework, the Microsoft Compact Framework [Mic05b], or the Mono Project [Mon06a] does not meet the requirements of limited resources of systems. There are few implementations of the CLI for small mobile devices e.g. for Symbian OS based [Gef05a], or for Linux based [Dot06a].

The memory footprint of an executable assembly is calculated by the assembly itself, the custom libraries used, the *Base Class Library* (BCL) and the *Common Language Runtime* (CLR). These are four items where size optimization can occur. In this paper the first three items were focused on. CLR optimization would harm the "compile once run everywhere" approach of CLI.

In this paper we present an approach to reduce the memory footprint of an executable assembly in that way the unused library functionality is not required to be present at runtime.

This can be achieved by compacting an assembly with its used library functionality into a self-contained assembly. The self-contained assembly will contain only required library functionality and will become smaller than the combined libraries. Furthermore the number of referenced assemblies which are required to be loaded is reduced to the self-contained assembly itself. The self-contained assembly is smaller than the sum of previously referenced assemblies.

This work is based on the PERWAPI [Gou05a] library, which is extended to the needs of creating self-contained assemblies.

The rest of this paper is structured as follows: Section 2 briefly reviews the Common Language Infrastructure. In Section 3 the mechanism of executing CIL-code is discussed in detail. Next, self-contained assemblies as approach for optimized memory footprints and predictable behavior in are presented in section 4. Section 5 gives an overview of related work followed by conclusions and future plans.

2. COMMON LANGUAGE INFRASTRUCTURE

The CLI standard specifies the executable format, a virtual runtime environment (*Virtual Execution System* (VES)) and a set of libraries as implemented in the Microsoft .NET Framework, Shared Source Common Language Infrastructure (SSCLI) [Mic02a], or in the Mono project.

CLI executables, called assemblies are encoded in the *Common Intermediate Language* (CIL) instruction set. An assembly is the deployment unit of the CLI and may consist of multiple files (modules). An assembly is loose coupled with the BCL and other assemblies in a way similar to native applications and shared libraries.

CIL is a stream of bytecodes similar to processor instructions. Most opcodes are one byte long, a few 2 bytes long and may have an optional parameter (up to 8 bytes long). Every method consists of a header, a body and a possible footer. To evaluate opcodes a stack is used. Bytecodes are located in the method body.

Metadata

Assemblies are equipped with metadata about references, type names, method names... Metadata are organized in a number of named streams. These streams are divided into 2 types: metadata heaps and metadata tables. For executing assemblies the following metadata tables are basically involved:

- *Assembly*: Assembly defined in the PE file.
- *AssemblyRef*: For execution required assemblies.
- *TypeRef*: Used types defined in external assemblies. Every type in this table refers its resolution scope that is located in the *AssemblyRef*-table for the relevant cases.
- *TypeDef*: Contains all types that are defined within an assembly.
- *Method*: All methods that are declared by types in *TypeDef*-table. Every row in the *Method*-table is owned by one and only one row in the *TypeDef*-table.
- *MemberRef*: All methods or fields of external defined types that are accessed within the assembly. There is merely a 'forward-pointer' from each row in the *TypeRef*-table.

References in metadata tables are tokens into table rows and heaps or relative virtual addresses within the assembly. Heaps are constant pools used for metadata and CIL code.

Costa and Rohou [Cos05a] show that metadata size varies from 40 percent up to 80 percent of the whole assembly size for representative set of programs. The metadata split 70 percent to 30 percent into constant pool (heaps) and tables. Section 3 will show that major parts of the *#String* are not required for executing

CIL code. For example textual descriptions of variables and properties are needed for reflection purposes only.

Version compatibility

To overcome the problem resulting from different versions of dynamic libraries on Windows systems [And00a] the CLI introduced a version management that builds up on version numbers and public keys. An assembly version number consists of four parts: major, minor, build and revision number. To make an assembly reference distinct the assembly must have a strong name. Strong names guarantee name uniqueness by relying on unique key pair. All shared assemblies that reside in the GAC must have a strong name. The BCL of actual CLI implementation have all the same standard public key that does not require a private key to sign. This is done to provide vendor independent execution of assemblies. That means an assembly which has references to the BCL (mscorlib.dll) may behave differently with different BCL implementations.

3. EXECUTION OF .NET ASSEMBLIES

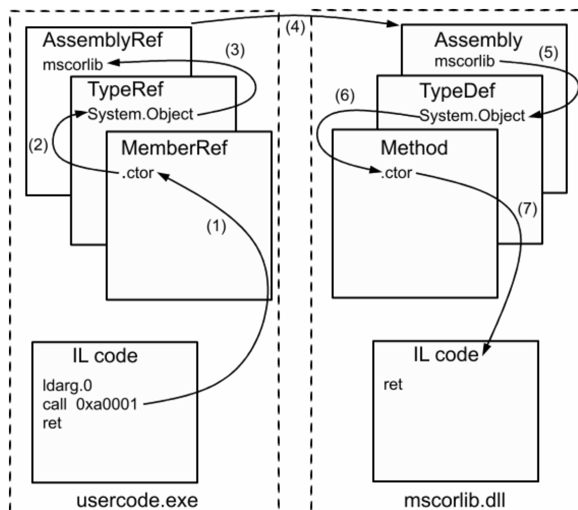

Figure 1: Resolving of an external method

When the CLR loads an assembly and starts executing a method all assemblies referenced within that method have to get loaded too. This means that all assemblies referenced in this assembly will be loaded, even though they might not be needed most of the time the application is executed.

A way to reduce the number of loaded modules is to merge multiple modules into one [Mic06a]. In terms of the CPU, assembly loads have fusion binding and CLR assembly-loading overhead in addition to the LoadLibrary call, so fewer modules mean less CPU time. In terms of memory usage, having fewer assemblies also means that the CLR will have less state to maintain.

To create the executable image the CLR has to locate referenced CIL code within an assembly. The complexity of this task is different for assembly internal and assembly external references. Figure 1 shows how CIL code of an external method will be located:

1. A CIL operation (call) has a token operand that points to a *MemberRef*-table row.
2. The *MemberRef*-table row contains the name of the method and a token into the *TypeRef*-table.
3. In the *TypeRef*-table row the namespace, the type name and a token into the *AssemblyRef*-table are included.
4. The *AssemblyRef*-table row provides the target assembly name and optional a version number as well as a public key token.
5. Within the referenced assembly the CLR looks into the *TypeDef*-table for the requested type. This is done by a linear search with string and signature comparison until the matching row is found.
6. The linear search for the matching method row in the *Method*-table is optimized in the way that the start of the relevant rows is known.
7. The matching *Method*-table row provides the address to CIL code within the PE-file.

This task must be repeated for every external method. In comparison with an external method call requires an internal method call a single lookup in the Method-table to get the address of the CIL code within the assembly. Recapitulating it has been reflected that loose coupling of assemblies and consequential external references cause the following drawbacks:

- *Memory consumption*: each external assembly must be loaded and metadata tables have to build up.
- *Processing power*: multiple indirections, linear search, string and signature compare during reference resolving cause additional CPU time in contrast with internal references.
- *Memory footprint*: combination of functionality into a single assembly (mscorlib.dll) causes a high CLR memory footprint if only a single type is referenced.
- *Revisable code*: CIL within an assembly can be inspected for validity. External assemblies especially the BCL may be implemented differently and makes it impossible to predict the behavior of CIL code.

These drawbacks can be minimized if all external referenced functionality is assembled to a single assembly. This harms the loose coupling, but it allows lower memory footprints and to analyze the assembly in terms of CIL code.

4. SELF-CONTAINED CLI ASSEMBLIES

A key feature of the CLI is the revisable bytecode-based execution of assemblies. The verification is done at runtime. But there are also needs for static revisable code before runtime e.g. prevent exceptions while runtime.

The loose coupling and dynamic linking of applications and libraries assemblies does not permit an static evaluation of CIL code, because CLRs may provide different implementations of relevant assemblies.

To overcome version conflicts of assemblies, CLI introduced strong names and side-by-side execution of different versions of the same assembly.

This works fine for most strong named assemblies, but fails for the BCL.

A static revisable assembly might not have dependencies to CLR-provided assemblies. With the self-contained assembly approach a static revisable format based on CIL code is proposed. This approach lifts up problems through different implementations of referenced assemblies.

Self-contained assembly features are:

- Minimal memory footprint
- Predictable behavior based on CIL-code
- Reduced startup time

The memory footprint of the runtime environment for an assembly is calculated by the CLR, the relevant libraries and the assembly itself. In general every assembly uses BCL features (e.g. `System.Object`). The BCL is represented as mscorlib.dll [Ecm02a]. But mscorlib.dll implementations of .NET Framework, Mono, Portable.NET [Dot06a] and Rotor provide different additional features, which are not used by most assemblies. Independently from the amount of mscorlib.dll features by an assembly the memory footprint for the BCL is fixed. Self-contained assemblies do not need additional library assemblies and form together with the CLR the minimal footprint for an execution environment. This feature targets mainly memory restricted systems.

Prediction of execution behavior of a self-contained assembly is possible, because all executable CIL codes are within the assembly. A static behavior evaluation can be done before runtime and allows for example prediction of memory consumption.

Dynamic linking of assemblies at load time causes delays until the first CIL code is executed. The time is needed for loading assemblies and resolving references. Self-contained assemblies does not require additional assemblies, therefore the startup time is shortened.

```
public class Hello{
  public static void Main(string[] args){
     Object obj=new Object();
     Console.WriteLine("Hello World!");
  }
}
```

Figure 2: Simple C# Hello world

Figure 2 shows a C# program cutout that has a Main-method where an instance of Object is created and "Hello World" is printed out. The second program in figure 3 shows the IL-code[1] of the Main-method generated by the Ildasm tool. The local variable `obj` disappeared, because it is not used furthermore. A instance of `System.Object` is created with a call of `.ctor()` from the mscorlib assembly. Then the string "Hello World" is printed out by an call of `System.Console::WriteLine` from the mscorlib too.

```
...
.method public hidebysig static void  Main(string[] args) cil
managed
{
 .entrypoint
 .maxstack  1
 newobj     instance void [mscorlib]System.Object::.ctor()
 pop
 ldstr      "Hello World!"
 call       void [mscorlib]System.Console::WriteLine(string)
 ret
}
...
```

Figure 3: IL code of the compiled Main-method

The program in figure 4 is generated from the second program where the `System.Object` type was included. The `System.Object::.ctor()` call does not leave the assembly scope. The rest of the program behaves the same.

The two IL-programs differ also in the .maxstack value, because the Microsoft C# compiler generates a Fat-method header and the PERWAPI library a Tiny-

[1] The C# source code was compiled with .NET Framework v1.1 compiler and optimization (/optimize+) enabled.

method header. None of the requirements for a Fat-header are satisfied, so the 1 byte Tiny header is a better alternative for size optimization.

```
....
method public hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  .maxstack  8
  newobj     instance void System.Object::.ctor()
  pop
  ldstr      "Hello World!"
  call       void [mscorlib]System.Console::WriteLine(string)
  ret
}
...
```

Figure 4: IL-code of Main-method with System.Object included

This demonstrates the adaptable level of containment for specific aims. The `System.Object` type was included and the reference to `System.Console ::WriteLine()` was kept.

Creating self-contained assemblies

Self-contained assemblies do not have any external references. This means a CLR should able to execute a self-contained assembly without loading the BCL or other managed assemblies.

In contrast to statically linked native binaries, the CLI abstracts from the operating system and the underlying hardware. This fact makes it feasible to build a CLR independent CLI assembly.

To get a self-contained assembly, the relevant assembly must be disengaged from type references to external assemblies. This work can be done by processing IL textual representation or by using an assembly manipulation library.

In this project the library approach is used, because ILDASM approach requires a lot of text substitution and depends on available CLI framework tools.

The Reflection API of the .NET Framework does not supports access to CIL code. Microsoft's new compiler framework Phoenix allows assembly modifications within a compiler run. After evaluation of capabilities of different assembly manipulation frameworks the work presented in this paper finally bases on PERWAPI [Gou05a] developed at the Queensland University of Technology. PERWAPI provides an abstract representation of the PE-file embodied as object oriented structure. The library is implemented in C# and is released as available for free. PERWAPI was extended to support the creation of self-contained assemblies.

Figure 5 shows the creation of self-contained assemblies with the Linker tool and an optional configuration. The assembly on the left side references the BCL (mscorlib) and may have references to multiple custom libraries.

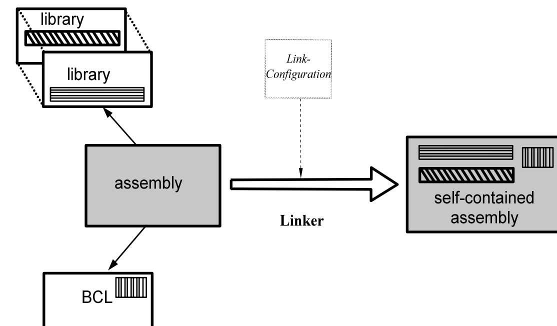

Figure 5: Creation of self-contained assemblies

The PERWAPI-based linker tool resolves references controlled by an optional configuration file. The configuration allows the instrumentation of the assembling process inside the linker tool. The source for a type to import could be set or types that should kept as references.

Every type defined in an assembly must be reviewed for the following list of elements:

Custom Attributes

A Custom Attribute points to a type constructor method and contains optional constructor values. Attributes can occur at assembly level, type level, and method level.

Type

A Type has a parent type except `System.Object` and may implement a number of interfaces. Methods describe operations that may be performed on that type. Fields are named subtypes of a type.

Interface

Interfaces are special types that do not have a super type and contain no CIL code.

Method

A Method is a named operation and is characterized by the types of its parameters. Besides the parameter types also the return type and possible Custom Attributes have to set to the resolved type. Local variables are unnamed subtypes within a method resolution scope. CIL code may have a type, method or field parameter. Exception clauses are defined by a code range and the type of the exception.

Event

Events are handled like fields of a type.

CIL code

The following types of IL codes must be checked for references to types, methods or fields references:

- Type Op.: castclass, newarr, initobj, ...

- Method Op.: call, calli, callvirt, newobj, ...
- Field Op.: ldfld, ldflda, stfld, stflda, ...

The challenge of assembling self-contained assemblies is to verify types for references and generate a consistent PE-file. The current version of self-contained assemblies addresses CLI v1.1 features only. There are further size optimizations practicable. To reduce the size of the constant pool, some kind of type descriptions can be shorten or eliminated. Custom type names not required by the CLR, except special names e.g. type constructor.

Proof of concept results

The current implementation of self-contained assemblies targets desktop CLR like .NET, Rotor, Mono or Portable.NET.

```
public static int Main(string[] args){
  Object obj=new Object();
  return 1;
}
```

The above C# program has a single external reference (`System.Object::.ctor`) in CIL representation. But for the self-contained version a second method from `System.Object` must be imported, because the CLR calls the destructor (`Finalize()`) of the CLI-base type without further reference.

The compiled[1] assembly with mscorlib reference had a size of 3072 bytes. The size of the CLR is not considered, because it assumed to be constant. So the memory footprint with .NET v1.1 mscorlib.dll is 2141184 bytes.

The self-contained version has an oval size of 2048 bytes and contains no references. These results are prestigious in no means, but the potential of self-contained assembly optimization.

To process more complex programs a clean BCL implementation is reasonable, because existing mscorlib.dll implementations are using none BCL features[2] for BCL functionality.

CLR implementation issues

The CLI defines a lot of possibilities for optimized CLR implementations. This section discusses these optimizations in terms of portability of self-contained assemblies among different CLR.

The CLR is responsible for resolving references to assemblies and loading types. References to external types are available in textual representation. CLI metadata are organized as a number of cross referenced tables. A referenced in type in an external assembly can have references to the same assembly or the external assemblies. The CLI suggests resolving all references before start the execution. Therefore all related assemblies must be loaded to create a consistent memory image.

For optimization issues the CLI introduced build in primitive types e.g. `bool`, `char`, `object`, `string`, `...`, which does not induce type references as long no type specific operation were performed.

In contrast to Java the CLI provides an internal mapping of primitive type to their wrapper types. The CLR knows the mapping of primitive types to their wrapper types e.g. `object≡System.Object`. The mapping of primitive types to BCL types, inside the CLR, is realized with string compare, because a type reference is given in textual representation. For types implemented inside a self-contained assembly this mapping is possible further on.

The CLI supports multiple ways to implement type methods. Possible implementation flags [Lid02a] for types inside the BCL:

- *cil*: The method is implemented in CIL code.
- *internalcall*: This flag indicates that the method is internal to the runtime and must be called in a special way.
- *runtime*: The method implementation is provided by the runtime itself.
- *pinvokeimpl*: The method has unmanaged implementation and is called through the platform invocation mechanism P/Invoke.

A *cil* implemented method can be executed by any CLR. An *internalcall* method is not portable among CLR implementations. This flag can occur in the BCL and additional features provided by the CLR. A *runtime* supplied implementation is also CLR dependent. The *pinvokeimpl* flag indicates the CLR provided mechanism (P/Invoke) to call native code. Figure 7 shows three different implementations of the System.Object::Equals(object) method.

The Microsoft .NET Framework uses the *internalcall* manner to perform the comparison. This implies the existence of a dispatch table for *internalcalls*.

```
Microsoft .NET v1.1.4322
.method public hidebysig newslot virtual instance bool
Equals(object obj) cil managed internalcall {}
```

[1] csc /optimize+ simple.cs

[2] Class attribute System.Runtime.InteropServices.ClassInterfaceAttribute::.ctor in .NET v1.1 System.Object implementation

```
Mono v1.1.13.2
.method public hidebysig newslot virtual instance bool Equals(object obj) cil managed
{
 .maxstack 8
 IL_0000: ldarg.0
 IL_0001: ldarg.1
 IL_0002: ceq
 IL_0004: ret
}
```

```
Compact Framework v1.0.500
.method public hidebysig newslot virtual instance bool Equals(object obj) cil managed
{
 .maxstack 8
 IL_0000: ldarg.0
 IL_0001: ldarg.1
 IL_0002: call bool System.PInvoke.EE::Object_Equals(object, object)
 IL_0007: ret
}
.method public hidebysig static pinvokeimpl("mscoree" as "#17" winapi) bool Object_Equals(object obj1, object obj2) cil managed preservesig {}
```

Figure 7: Implementation of System.Object:: Equals(object) in .NET, Mono and Compact Framework

Mono provides a implementation based on CIL code, which makes the implementation portable.

In the Compact Framework BCL `System.Object ::Equals(object)` is implemented with a additional call through the P/Invoke mechanism.

The current version of self-contained assembly's implementation is portable among different CLR as long as no implementation specifics are used. One can benefit from self-contained features as long as is executed with the CLR that provided the BCL implementation.

5. RELATED WORK

There are several approaches to optimize Java class files to meet the requirements of small embedded devices. The optimizations are often done on a per class basis.

IBM's WebSphere® Studio Device Developer (WSDD) [IBM06a] includes the SmartLinker tool (formerly JAX [alp06a]) to optimize J2ME [Sun06a] applications.
SmartLinker removes unused code, merges classes, and introduces short identifiers to reduce the overall code size. Resulting applications are composed in the Java Executable format (JXE), which is not interoperable with jad/jar format as specified in J2ME.

Rayside et al. [Ray99a] propose a modified Java class file format with significant space reduction with little or no runtime penalty.

Clausen et al. [Cla00a] use macros for multiple occurrences of code fragments and an extended JVM with macro support.

The JamaicaVM[aic06a] developed by aicas GmbH includes a builder tool for integrating Java bytecode and a corresponding Virtual Machine implementation into a single executable application binary. Bytecode is embedded as C-Array definition and linked with the JamaicaVM library.

TinyVM[Sol06a] is a firmware replacement for the Lego™ Mindstorm™ RCX hardware. The firmware executes (interprets) Java programs that are compacted into custom images.

The Lego.NET [Osm05a] project has developed a GCC front-end which translates CIL code into native machine code of the Lego™ Mindstorm™ RCX processor.

Microsoft's .NET Compact Framework is a subset of the .NET platform for mobile and embedded devices. The Compact Framework class libraries occupy at least 2 Megabyte of memory. The assembly format and execution environment differ only in trifles from the desktop version.

Microsoft's ILMerge[Mic06a] is a utility that can be used to merge multiple .NET assemblies into a single assembly. ILMerge does not support a selection of types which should be merged together.

AppForge, Inc. offers with Crossfire[App06a] a product for multi-platform applications for mobile and wireless devices based on .NET. The CIL byte-code is transferred into a custom executable format that is executed by platform specific Crossfire-Client software.

6. CONCLUSION AND FUTURE WORK

This paper proposes an approach of self-contained assemblies to reduce memory consumption and shorter startuptime while executing the assembly. CLI assemblies are loose coupled with other assemblies (shared class libraries, custom libraries).

Creating of self-contained assemblies is done at type level with a customized version of the PERWAPI assembly manipulation library. The compaction of assemblies bases on referenced types of an assembly and requires no source code, nor compiler support. Self-contained assemblies are size optimized in terms of assembly footprint and memory consumption while execution.

Furthermore the effect of an executed self-contained assembly is identical among the acceptance the CLR is CLI-complaint and no CIL-code is executed outside of the assembly.

The customized PERWAPI library allows adaptive compaction at type level that means certain types remain as references.

It has to be analyzed to what extent the abstraction of CLR internals from the BCL implementation could be realized CLI-compliant.

The proof-of-concept results must be analyzed in terms of memory consumption, startup time and execution performance with CLR implementations.

Self-contained assemblies could offer useful features for embedded systems development, for predictable execution behavior and more generally for an adaptive deployment format.

7. ACKNOWLEDGMENTS

We would like to thank the reviewers for their useful comments and suggestions.

8. REFERENCES

[Aic06a] aicas GmbH. JamiacaVM. Available at *aicas.com*, 2006

[And00a] Anderson R. The End of DLL Hell. Microsoft Cooperation. Available at *msdn.microsoft.com/library/en-us/dnsetup/html/dlldanger1.asp*, 2000

[App06a] AppForge, Inc. Crossfire homepage. Available at *www.appforge.com/products/ crossfire*, 2006.

[Cla00a] Clausen L.R., Schultz U.P., Consel C., and Muller G. Java bytecode compression for low-end embedded systems. ACM Transactions on Programming Languages and Systems, 22(3):pp.471–489, 2000.

[Cos05a] Costa R.,and Rohou E. Comparing the size of .net applications with native code. in CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pp. 99–104, ACM Press, 2005.

[Dot06a] The DotGNU project. Portable.NET. Available at www.dotgnu.org, 2006

[Ecm02a] Ecma international. Standard Ecma-335, Common language infrastructure (Cli). Available at *www.ecma-international.org/ publications/ standards/Ecma-335.htm*, 2002.

[Gef05a] Gefflaut A., van Megen F., Siegemund F., Sugar R. Porting the .NET Compact Framework to Symbian Phones – A Feasibility Assessment. .NET Technologies'05 conference proceedings, UNION Agency – Science Press, ISBN 80-86943-01-1, 2005

[Gou05a] Gough J., and Corney D. PERWAPI-a pe file reader/writer. Available at *www.plas.fit.qut.edu. au/perwapi*, 2005.

[IBM06a] IBM. WebSphere Everyplace Micro Environment. Available at *www-306.ibm.com/ software/wireless/wsdd*, 2006

[Int03a] International Standards Organisation. Informationtechnology – Common Language Infrastructure, ISO/IEC 23271:2003(E) First edition, 2003.

[alp06a] alphaWorks/IBM. JAX. Available at www.alphaworks.ibm.com/tech/JAX, 2006

[Lid02a] Lidin S. Inside Microsoft .net il assembler.Microsoft Press, 2002.

[Mic02a] Microsoft Corporation. Shared source common language infrastructure. Available at *msdn.microsoft.com/net/sscli*, 2002.

[Mic05a] Microsoft Corporation. .NET Framework. Available at *msdn.microsoft.com/netframework*, 2005.

[Mic05b] Microsoft Corporation. .NET Compact Framework. Available at *msdn.microsoft.com/ netframework/programming/netcf*, 2005.

[Mic06a] Microsoft Research. ILMerge, Available at *research.microsoft.com/~mbarnett/ILMerge.aspx*, 2006

[Mon06a] The Mono project. website. Available at *www.mono-project.com*, 2006.

[Osm05a] Operating systems and middleware group. Lego.net website. Available at *www.dcl.hpi.unipotsdam.de/research/lego.NET/*, 2005.

[Ray99a] Rayside D., Mamas E., and Hons E. Compact java binaries for embedded systems. In CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, page 9. IBM Press, 1999.

[Sol06a] Solorzano J.H. TinyVM website. Available at *tinyvm.sf.net*, 2006.

[Sun06a] Sun Microsystems, Inc. Java Platform, Micro Edition. Available at *javasoft.com/j2me*, 2006