

Application platforms for embedded systems: Suitability of J2ME and .NET Compact Framework

Koen Victor

Computer Science Department
Katholieke Universiteit Leuven
B-3001 Leuven, Belgium

Yves Vandewoude

Computer Science Department
Katholieke Universiteit Leuven
B-3001 Leuven, Belgium

Yolande Berbers

Computer Science Department
Katholieke Universiteit Leuven
B-3001 Leuven, Belgium

Keywords

Java Micro Edition, .NET Compact Framework, Embedded System, software reuse

ABSTRACT

We compare Java 2 Micro Edition (J2ME) and the .NET Compact Framework as programming platforms for complex software on small devices. In addition, specific problems of the two platforms with regard to embedded systems (such as hardware addressing) are discussed. As a base for our comparison, we have ported a relatively large component middleware system to both platforms. Our research shows non trivial differences between the platforms. A performance comparison shows that both J2ME and the .NET Compact Framework exhibit similar performance characteristics with a small speed advantage for .NET. A much more important fact, however, is that on both platforms the usage of platform specific concepts yields considerable performance gains.

1. INTRODUCTION

Microsoft .NET Compact Framework (NCF) and Java 2 Platform Micro Edition (J2ME) offer a development platform for applications on tiny devices, based on their full blown counterparts in the desktop and server world. A considerable amount of PDAs and cell phones are already equipped with at least one of these platforms. The purpose of this paper is to make a comparison of these platforms. In the process, their functionality and benefit for the programming task is compared to their counterparts in the desktop world.

More than the embedded market, the desktop application market has known several popular application platforms that support modern application development techniques to enable reuse, convenient adaptation, and rapid development. As a range of programs in the desktop world are now becoming available for embedded devices too, it is interesting to see how the scaled down versions of these application platforms are meeting the needs of embedded programming. The requirements of embedded programs, for example relating to available memory and computing power are still very specific and need to be dealt with appropriately.

2. METHODOLOGY

The Java platform and the .NET (Compact) Framework are an attempt to fulfill the current needs of the soft-

ware world. On all of these platforms, solutions are put forward for the same problems related to compactness, performance and flexibility of the software. The way of solving these problems differs and it is interesting to see how these differences influence the design of a program.

A good way to compare between platforms is to port a program from one platform to the other platform. A port takes more time than a straightforward API comparison, but gives much better and more thorough results as not all subtleties are clear-cut from an API documentation. Additionally, some differences only manifest themselves in real life situations. Porting an existing program has the advantage that no program has to be developed from scratch.

Several criteria apply for the program to be ported: (1) the program should be a useful application in both the desktop and the embedded world, (2) it should use a big part of the relevant API of the platform and (3) it should make it possible to stress the platform to its limits. With these criteria in mind, Draco (section 3), a J2SE middleware system developed and actively maintained at the Department of Computer Science of KULeuven was selected. The rewrite should not be seen as a 'literal' translation; for concepts without a natural alternative on the .NET platform, the common .NET solution is used. The result is a program that is designed and implemented as it would have been developed for .NET in the first place. For the purposes of this paper, the choice to start from a J2SE program is arbitrary.

The Java and .NET Framework version of Draco are then adapted to work on the embedded variant of the platforms, J2ME and the .NET Compact Framework. As such, the conceptual and practical differences between the platforms become clear. The rationale behind starting from a program on a desktop platform is related to the incentive of this paper: many applications start out their life in the desktop world, to prosper as programs on limited devices later on.

3. DRACO

Draco is a middleware platform for component based development implemented in J2SE. For an elaborate description of the architecture of Draco, please consult [1].

The architecture of Draco consists of five core elements;

the component manager, the scheduler, the message manager; the connector manager and the module manager. In a distributed context, each node will run the entire Draco system consisting of the five core units. In the implementation, platform specific functionality is used. For example, the scheduler uses advanced locking and thread manipulation. The message manager uses nanoxml, a lightweight XML parser for Java. The implementation of the module manager and the component manager uses the Java event system and the Java Class Loader mechanism. To deliver a message between components, there is extensive use of reflection. The Draco source code size is 1395 lines.

4. PORTING DRACO

Firstly, the port from J2SE to J2ME is discussed. Secondly, we give an overview of the translation of the J2SE code of Draco to the .NET Framework. Then we present the port of the .NET Framework version to the .NET Compact Framework version.

As the device, we chose a Compaq PocketPC Ipaq H8370 with Microsoft Windows CE 4.20. It is equipped with an Intel StrongARM 206 MHz CPU, 64 MB RAM and a 240x320 pixel display. This is not a state-of-the-art device, but one could expect to see desktop-like applications on it.

4.1 Adaptations from J2SE to J2ME

There are several J2ME virtual machine implementations. IBM's J9, NSICOM's CrEme JVM and Esmertec's JBed are well known. We opted for the IBM J9 JVM, for which a trial version is available. This trial version comes with an integrated development environment for easy deployment on the PocketPC. The choice of the JVM only has an influence on the performance of the system, not on its functionality because this is specified by the J2ME configuration and profile combination.

The effort required to port a J2SE application to J2ME greatly depends on the chosen configuration and profile. As the *Limited Connected Device Configuration* does not provide reflection, this is not an option. For the *Connected Device Configuration*, three profiles can be selected. The *Foundation profile* has no support for the Abstract Windowing Toolkit (AWT) and Draco has an AWT based GUI. The *Personal Basis Profile* has limited AWT support, but does not support the AWT events used in the Draco GUI. The GUI can be rewritten however, to work with this profile if the virtual machine footprint is really an issue. Most desktop-like applications have an advanced GUI. Therefore, we chose the *Personal Profile*, which is the most 'complete' profile of all.

With the Personal Profile, no adaptations of the source code were necessary. This provides a very convenient way to make a compact program from a desktop program. The functionality that is not supported by the Personal Profile, such as RMI (Remote Method Invocation), is not used in mainstream applications.

4.2 Translation from Java to .NET

The basic language features of Java can be translated to C# and a substantial part of the Java Class Library has a worthy alternative in the .NET Framework. To translate these features, Microsoft offers the Java Language Conversion Assistant (JLCA). When we applied the JLCA 2.0 on Draco, substantial parts of the code were translated, but it still took a considerable amount of effort to make the program compile and run. For example the Java Class Loader is too different from the loading mechanism of assemblies in .NET to be converted by the JLCA (section 5.2.2). The same remark holds for native methods in Java (JNI) (section 5.1.2). For Java-specific libraries one has to provide an alternative manually. For example in the Draco scheduler, threads are put in a Java ThreadGroup to be able to make a common modification on them. In .NET, there is no support for a ThreadGroup.

The JLCA conversions are not always useful. For example, in Java it is allowed to return an object of a class that is only accessible in the package, in a protected method. A protected method allows access from the same package or subclasses. In C# this is not useful because methods outside the assembly cannot do anything with the object, but the JLCA produces such code. For the external libraries like *nanoxml*, no alternative is presented. The conversion of the Draco GUI was very limited.

4.3 Adaptations from .NET to NCF

.NET Compact is a 'one size fits all solution' that is not configurable to the target device or application. We had to make a considerable effort to make Draco run on the .NET Compact Framework. Most modifications are small, but there are many. In the .NET Compact Framework, the methods provided are less flexible; more work is needed to get arguments in a supported format. For Draco, the most notable thing missing is the lack of the synchronization methods `Monitor.Wait(object)` and `Monitor.Pulse(object)`. These synchronization primitives were implemented using `P/Invoke`.

5. COMPARING IMPORTANT CONCEPTS

In the next section, we discuss how well the two embedded platforms fulfill the requirements of embedded applications by comparing the ported program Draco on both embedded platforms. Programs for embedded systems and tiny devices differentiate themselves from desktop programs by the following characteristics:

Access to exotic hardware: Some embedded systems need to address special purpose hardware using C/C++ device drivers. Access to all the device driver features should be possible without too much overhead or latency.

Low memory usage: The platform should either allow the programmer to do efficient memory management or else it should do it automatically.

Modularization of code: Code modularization and deferred loading may improve memory requirements.

Compactness of code: A key consideration for tiny systems with little storage is the memory footprint of an application and the runtime environment (including all libraries).

Power Consumption: The typical user of PDAs, the GSM or other easily portable devices is very mobile and has power provided only over limited periods of time. An application platform should allow the management of resource intensive parts of the hardware such as the screen or the CPU.

Good performance on slow systems

5.1 Access to exotic hardware

The PocketPC has an accurate hardware clock that can be used to do precise time measurements. This clock is recognized by the operating system Windows CE, and can be accessed via the library `coredll.dll`. The built-in in timing features of both embedded platforms use this clock for timing measurements, but the precision of these methods is limited to 1ms. Like all operating system libraries on Windows CE, it is written in C. To see how well external libraries can be used, we use `coredll.dll` to access the hardware clock;

`QueryPerformanceFrequency(LARGE_INTEGER ← *lpFrequency)` returns the frequency of the hardware clock, if one is available.

`QueryPerformanceCounter(LARGE_INTEGER ← *lpPerformanceCount)` returns the number of clock ticks since starting the device.

We will show that the clock itself is more precise, and can be used with an accuracy up to 0,005 ms. This clock will be used for the performance evaluation in section 6.

5.1.1 Accessing the hardware clock in J2ME

To have access to an external library in Java, the *Java Native Interface* (JNI) can be used. The JNI allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly. The reverse is also possible; the *Invocation API* allows to embed the Java Virtual Machine into native applications. To use JNI, the external methods are declared in the Java code. The Java program is compiled and a header file is generated with `javah -jni`. Access to the external method is primarily done by implementing the header file. Then the header and the implementation file(s) can be compiled and the Java program can be run.

JNI provides easy translation between the primitive types of C and Java. It also allows to access member variables, handle exceptions, program callback methods and use basic synchronization. JNI provides a fairly complete environment to access external libraries, but for calling a simple method JNI requires some effort that could have been built into the Java language itself.

5.1.2 Accessing the hardware clock in NCF

In .NET, the way external libraries can be used differs depending on the language in which these are written

and the language of the .NET project in which access to the library is wanted. Following is a list of the possible use cases.

The library is a .NET library: A .NET library (*assembly*) can be used by any language that is supported in .NET. This is discussed further in section 5.2.2.

The library is written in a language supported in the .NET Framework and the code is public: Apart from the support of languages as C#, VB.NET and J#, compilers or communication techniques have been developed to provide interoperability for other languages. For certain languages a variant has been developed, for example MC++ as a variant of C++.

The library is a C Dynamic Link Library (DLL): The .NET Framework has support for calling C DLLs by the Platform/Invoke interface. This possibility is also applicable on C++ DLLs with a public C interface.

The library is a C++ DLL: One of the design goals of Managed Extensions for C++ was that C++ DLLs would be accessible from a MC++ class. This way, one could use methods from the library in a .NET assembly, and if needed one could export these methods as .NET methods for other .NET projects. This is not possible on .NET Compact Framework, because MC++ is not supported. We encountered some problems with this approach on the .NET Framework, as we will explain in the case study in section 7.3.

The other cases...: If none of the above options is possible, in many cases it is possible to write a C interface that calls the methods in the library. C and C++ have good operability with many languages, by means of *wrappers* or adapted compilers. This C interface can be used via P/Invoke in .NET.

For accessing the hardware clock, we used the P/Invoke method. P/Invoke is often used to interact with Win32 libraries and is supported by the CLR. To use P/Invoke in C# on .NET, the prototypes of the methods are 'imported' in the .NET code using the `DLLImport` attribute. For the conversion between parameter types of the DLL and the .NET types (*marshaling*), the functionality in `System.Runtime.InteropServices` can be used. Much if not all of the conversion functions necessary to use Win32 libraries are there, but for external DLLs with callback functions and parameters that are difficult to convert to .NET types, it may be easier to implement a C wrapper that does the conversion.

5.1.3 Latency

Although neither J2ME and the .NET Compact Framework are suitable for real-time applications, it is interesting to see how quickly hardware can be addressed through the interfaces of these platforms. To measure this, we can call the Start and Stop methods of the clock implementation. To level out the JIT compilation effects, we execute the methods 20 times and take an average over the last 19 calls.

Table 1: Latency of the clock implementation

	first execution (ms)	average (ms)
J2ME	0,160	0,025
.NET Compact	1,697	0,005

From table 1, we learn that the overhead for calling an external library on J2ME is less than 0,025 ms. On .NET the overhead is less than 0,005 ms. In comparison to .NET, the first call of the clock in J2ME is more than 10 times faster. The subsequent method calls are up to 5 times slower than those in .NET. It seems that the .NET Compact Framework needs more time to do the JIT compilation of the clock methods and to initialize the access to the library but is faster after that. These are very satisfying results that allow for accurate performance measurements.

5.2 Modularization of code

Draco allows to load components at runtime by specifying their name. If the Draco runtime finds the component in its searchpath, it gets loaded. Also modules for extending the runtime itself and the commands for the command line interfaces are loaded this way. In Java, this is implemented by using the Java class loader. Every component or module resides in its own JAR-file. In the .NET implementation, every component resides in its own .NET assembly. The following sections explain the differences in using these loading mechanisms.

5.2.1 Loading Java libraries

To load a class it is sufficient if the class is located in the searchpath of the Java environment. The class loader searches for a class and loads it. Every class has a reference to the class loader by which it was loaded. If the class is found, it gets loaded. Otherwise a *ClassNotFoundException* exception is thrown.

For ease of deployment, every component has its own interfaces, for example *Component*, that is exactly the same as the interfaces in the JAR with the Draco runtime. This is not possible in the .NET version. The .NET type checking is more stringent than the checking in Java. To make this clear, we provide an example of a possible erroneous cast, that can not happen in .NET.

In the following example, class A implements interface I (code snippet 1), class B implements another interface with the same name I (code snippet 2) but with a slightly different implementation. Figure 1 shows the physical location of the files.

Listing 1: Java class A implements interface I

```
public interface I {
    public String txt = "1st interface I";
}

public class A implements I {
    public A () {
        System.out.println("this is A");
    }
}
```

Listing 2: Java class B implements interface I

```
public interface I {
```

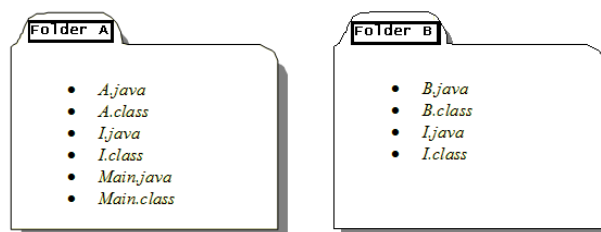


Figure 1: Physical layout of the files for the class cast example

```
public String txt = "2nd interface I";

public class B implements I {
    public B () {
        System.out.println("this is B");
    }
}
```

Listing 3 shows that in this case class B, that complies with interface I in the same JAR, can be casted to another interface I with another implementation.

Listing 3: erroneous cast

```
public class Main {
    public static void main(String[] args) {
        ClassLoader cl = ClassLoader. ←
            getSystemClassLoader();
        try {
            Class BClass = cl.loadClass("../B/B");
            System.out.println("B loaded");
            Object b = BClass.newInstance();
            System.out.println("instance B made");
            I interIB = (I)b; //cast to 1st I
            System.out.println("B casted to I");
            System.out.println(interIB.txt);
        } catch (Exception io) {
            System.out.println("Error..");
        }
    }
}
```

The output of the program is:

```
B loaded
this is B
instance B made
B casted to I
1st interface I
```

This is not an issue for Draco, but the form of deployment makes use of this behavior. This is not possible in .NET.

5.2.2 Loading .NET libraries

The .NET (Compact) Framework has a more complex mechanism to load .NET libraries at runtime. Microsoft implemented a rigorous versioning strategy in the .NET Framework, that enables applications to remain isolated from others and execute components 'side-by-side'. Assemblies have a manifest that describes the identity (name, version, ...) of an assembly. The version of a type (for example a class) is equal to the version of the assembly in which the type resides. Loading an assembly can be compared to using a class loader in Java. There are a few differences in the usage of the acquired type, caused by the measures to avoid the DLL versioning problems.

If an object of a loaded class is casted to a class that is not in the loaded assembly, or is not used in it, the cast can not succeed, even though the rest of the typing information is identical. If one wants to load a class and cast it to an internal interface, the loaded class should be compiled against the same internal interface. As mentioned in section 5.2.1, the implication for Draco is that it is not possible to load a component that is compiled against another *Component* interface, as is the case in Java. All components should be compiled against the Draco runtime that has the Component interface. In practice, this is not a problem.

Another difference with the Java class loading mechanism is that there are different methods to load assemblies, depending on the intended use of the library. These methods, that seem to do the same, act different in a way that is not clear from the documentation. Dependent on what method is used, the binding of an assembly happens in another ‘context’. There are three such contexts:

Load context is used if the assembly is found in the *Global Assembly Cache* (GAC) - this is not the case in Draco as private assemblies are used - or in the *ApplicationBase* path. This means that the assembly is found in the *obj/Debug* of the Draco project and `Assembly.Load("thed11")` is used. The assembly is put in the *static cache*.

LoadFrom context is used if an exact path is given to load the assembly (for example when `LoadFrom`, `CreateInstanceFrom`, but not `LoadFile`). The assembly is put in the *path based cache*.

‘neither’ context is used if the user and not the .NET assembly loader generated the assembly (for example `Assembly.Load(byte[])`, `Assembly.LoadFile`, or if reflection `Emit` is used.

Relevant for Draco is that when `Assembly.Load` is used, the dependencies (for example an interface) in another context are not available. This can be resolved by using the `AppDomain.AssemblyResolve` event that can notify a listener when a new assembly gets loaded.

If `LoadFrom` is used, then even a cast in the loaded assembly to an interface or class that is defined in that assembly is not possible and a *IOException* gets thrown because that assembly could not be found. For clarity: this is thrown from within code in that assembly! The cause of this is the use of two separate caches. For type-casting, the path based cache is not used, probably to make sure that a type of a path based assembly is not casted to a static assembly. Casts in a path based assembly are not taken into account, and this is at least questionable behavior.

Another consequence of this behaviour becomes clear if an assembly loads a class from itself. This is an alternative for working with several assemblies like described above. Draco is obtainable in the form of one JAR file with all commands. This distribution format should also be possible in the .NET version. To achieve this with

Table 2: Memory requirements for Draco

	J2ME (MB)	.NET Compact (MB)
VM footprint	1,55	1,56
Draco footprint	0,173	0,203
RAM usage	3,08	2,57

Draco, we compiled Draco as a DLL instead of an exe, and a separate program calls the entrypoint in the DLL. This is not possible if `LoadFrom` is used. Though the path to the assembly that contains the ‘external’ classes is the same as the path to the assembly that contains the interfaces (it is the same assembly...), the objects of the loaded types can not be casted to the common interface, because the path based cache is not looked at. We solved this problem using `Assembly.Load` and placing the components in the *ApplicationBase* path, because then components reside in the static cache.

5.3 Memory requirements

The memory requirements for Draco are listed in table 2. The footprints of the final Draco systems are almost the same, respectively 173KB and 203KB. The J2ME VM together with the Java class library takes 1,55 MB. The .NET Compact Framework needs 1,56MB. These are very reasonable numbers for the target devices, that have typically more than 32 MB of storage.

The memory requirements of the running implementations are comparable. The J2ME application takes 3,08MB of memory. The .NET Compact version needs 2,57MB. Both platforms do not support explicit freeing of memory. Even if an object is finalized, the memory may only be reclaimed when the Garbage Collector (GC) is activated. It is possible to call the GC manually (`.NET: System.GC.Collect()`, `Java: Runtime.getRuntime().gc()`) but there is no guarantee that the GC gets executed. Moreover, calling the GC manually is an effective way to slow down an application —the programmer often has a less clear view on what happens during the execution of a program than the GC itself. As a general sidenote we mention that using a virtual machine does not necessarily imply a code size reduction compared to native code [2].

5.4 Power Consumption

J2ME and .NET Compact do not provide any features through the API to regulate the power usage of the device. For J2ME, platform independence may make it less intuitive to provide features to, for example, dim the screen of a device. For .NET Compact, that is run on Windows CE, one would expect to see this kind of features. The intermediate code does not provide any support for power management. All code gets executed ‘at full speed’ and it is not possible to differ tasks to a moment when the device is not used by other programs.

6. PERFORMANCE COMPARISON

Both J2ME and the .NET Compact Framework collect general statistics during the execution of a program. These statistics give only a general overview of the actions of the program, like the number of method calls or

	<i>use case</i>	J9 (ms)	CLR (ms)
1	initialization VM 1st	2569,805	1101,805
	initialization VM 2nd	1989,640	1096,975
2	start of Draco, no GUI	4042,720	1532,500
3	start of Draco, GUI	4488,105	3800,205
4	load a dummy comp 1st	380,950	320,500
	load a dummy comp >1st	330,250	299,920
5	load a component 1st	410,875	378,400
	load a component >1st	350,560	330,420
6	connect components 1st	398,120	340,400
	connect components >1st	355,560	305,420
7	send a message 1st	270,120	240,400
	send a message >1st	268,100	240,410

Table 3: time measurements

invocations of the GC. Therefore, we use the clock explained in section 5.2 to obtain accurate time measurements. When measuring time on these platforms, it is important to keep the inherent determinism of virtual machines in mind.

First, not all classes are loaded immediately at startup time. It is possible that the class is loaded when it is used the first time, for example when creating an object. Of course this slows down the creation of the first object. Second, JIT compilation slows down the first invocation of a method. Third, the garbage collector has a big impact on the execution speed of a program, because during the execution all other threads are stopped. To guarantee the accuracy of a measurement, the possible effects of the code adaptation to call the clock should be accounted for. Also, it is important that the clock implementation is as short as possible and is used consequently in the same way.

6.1 Methodology

We compare seven relevant scenarios: (1) the start and initialization of the virtual machine, (2&3) the start of Draco with and without GUI, (4) the loading of a dummy component that does not register itself in Draco, (5) the loading and instantiation of a small number generator Draco component, (6) connecting the number generator to a number display component and (7) the start of the component Numbergenerator.

Before every measurement, the Start and Stop method of the clock is used at least once to make sure the code of the clock is compiled before the actual measurement. The test is executed with no other user applications running, and with enough memory to avoid effects of the garbage collector. For use cases like the loading of a component, the code path is executed 10 times and an average is made over the last 9 iterations, to make the influence of JIT effects visible. We subtracted the average clock latency as described in table 1 of section 5.1.

6.2 Discussion and interpretation

The measurement of the start-up time of the virtual machine is a special case because the timer we implemented can not be used. The initialization of the VM is considered to be started right after the start of the VM, and

finished when the static start method of Draco gets executed. The start time of the VM has is measured by a *native* PocketPC program. This program executes a first time measurement, and starts Draco. The second measurement is in the start method of Draco. The time measurement itself cannot be made exact: the time the operating system spends to call the virtual machine is also measured, but as this deviation is for both platforms, the values can be compared. The initialization of the VM is measured two times; the first time the VM is started after a *reset* of the device. The second time it is executed after stopping the first VM instance. This measurement is influenced by the use of shared system libraries by the VM and the caching of the operating system.

The results are presented in table 3. These measurements show in most cases a performance advantage for the .NET Compact Framework implementation. However, we find that the performance gain obtained by porting the application to .NET is not convincing enough to decide the better platform for desktop like applications on tiny devices.

7. CASE STUDY: A COMPONENT CAMERA APPLICATION

This case study allows to further compare the developing platforms, specifically relating to the networking capabilities, addressing of rare hardware and manipulation of images. The application consist of the Draco middleware system on a desktop system capturing images from a Sony DFW-VL500 camera [3]. These images are transported to the PocketPC using a network connection. On the PocketPC, the embedded version of Draco is installed. This Draco instance receives the images and displays them on the PocketPC screen.

7.1 Networking the PocketPC

The PocketPC is connected to the desktop computer using the Microsoft ActiveSync protocol. ActiveSync creates a virtual network between the PocketPC and the PC and connects them transparently of the actual physical connection. The PocketPC can address the PC using the regular networking facilities of Windows CE. As a result, both J2ME and the .NET Compact Framework can make a regular network connection to the desktop PC. An alternative for ActiveSync is Palm Hotsync Software.

7.2 Controlling the camera in J2SE

The control of the camera in J2SE is implemented using JNI, similar to accessing the hardware clock (section 5.1).

7.3 Controlling the camera in .NET

In a first attempt to control the camera using a C++ driver library, we followed the approach to use Managed C++. MC++ is C++ with the addition of keywords and attributes for the integration in .NET. MC++ has some limitations in comparison to C++, but it can use the .NET libraries and lets the garbage collector manage memory. The reuse of (legacy) C++ code on the .NET platform is not an easy undertaking. For example, multiple inheritance is not described in the Microsoft Common Lan-

guage Specification (CLS) and as such it is not available MC++.

A standard C++ class can be compiled for the CLR by using the C++ .NET compiler with the `/clr` switch. This code gets executed by the CLR, while retaining the functionality of C++. This class does not benefit from the .NET features, like garbage collection. Classes that are compiled this way cannot be used from other .NET languages. A 'managed' class is declared by the `_gcc` keyword and has a reference to `mscorlib.dll`. Objects of classes that are declared that way, are managed by the garbage collector. To other .NET classes, these classes behave like usual .NET classes; it is possible to instantiate and inherit from them. Using MC++, it is possible to call in a regular C++ class from within a managed method, and expose the result to other .NET methods. This is ideal for making existing C++ libraries available on the .NET framework without using COM or P/Invoke.

However, when we applied this technique to the camera library, we got a linker tools warning: `Linker Tools Warning LNK4210: unresolved external symbol X`, with X the symbols that are present in the CMU 1394 library. This turned out to be an architectural design flaw in the current .NET Framework [4]. We will not go into the specific details, but the root of this problem is that the algorithm to load mixed mode DLLs and make managed code work together with x86 code may result in a deadlock situation with certain libraries that define their own `DLLMain()`. It is possible to circumvent the problem partly by initializing the external library manually, but for the camera library this would be troublesome and it would not solve the problem entirely.

Therefore, a C++ wrapper was written that exports methods through a C-interface. These methods are accessed in .NET using P/Invoke, as described in section 5.1.

7.4 Server/Client

Both J2ME and the NCF have good support for networking. J2ME lacks the asynchronous network calls of NCF but this can be implemented using separate worker threads. We found that on the .NET Compact Framework, the TCP/IP connection via `ActiveSync` could not be closed, and was only closed after a timeout of approximately 4 minutes on the desktop side.

7.5 Image manipulation

The camera produces RGB images. To send these over the slow network connection to the PocketPC, we compress the images to the JPG format. In Java this is done by the Java Advanced Imaging API (JAI). In .NET, the `System.Drawing.Imaging` namespace is used. On the PocketPC, the image has to be converted to a regular 'bitmap' to be able to show it on the screen. In the NCF, the 'straightforward' implementation is shown in listing 4. This method suffers from serious performance drawbacks; the algorithm was not able to convert all images coming in over the slow network connection.

Listing 4: RGB to Bitmap: slow version

```
Bitmap result = new Bitmap(width, height);
int t1, t2, t3;
for (int j=0; j<height; j++){
    int k = j*width*3;

    for (int i=0; i<width*3; i+=3){
        t1 = image[i+k] & 0xFF;
        t2 = image[i+1+k] & 0xFF;
        t3 = image[i+2+k] & 0xFF;

        result.SetPixel(i/3, j, Color.FromArgb(↔)
            (255 << 24 | t1 << 16 | t2 << 8 | t3));}
    }
```

The bad performance of this method is due to the internal implementation of `SetPixel(..)` and `GetPixel(..)`. For example, each call to `SetPixel(..)` locks the `Bitmap result` for reading and writing. A more efficient method (listing 5) circumvents this locking by using a `BitmapData` object and *unsafe* code. The use of pointer arithmetic and the avoidance of the locking per pixel drastically improved the conversion speed up to 10 times for images around 40KB. In J2ME, the conversion in listing 4 is done directly on the `bytestream` array, without the need for specific imaging APIs.

Listing 5: RGB to Bitmap: optimized version

```
Bitmap result = new Bitmap(width, height, ↔
    PixelFormat.Format24bppRgb);
BitmapData bmd = result.LockBits(new Rectangle(
    0, 0, result.Width, result.Height), ↔
    ImageLockMode.WriteOnly, ↔
    PixelFormat.Format24bppRgb);

int bytesPerWidth = bmd.Stride;
int height = bmd.Height;
int width = bmd.Width;

byte* element = (byte*) bmd.Scan0+bytesPerWidth;

int k = 0;
for (int y=0; y<height; y++){
    int l=k;
    for (int x=0; x<width; x++){ // RGB is BGR
        *(element+2) = image[l++];
        *(element+1) = image[l++];
        *element = image[l];
        element = (byte*) element+3;
    }
    element += bytesPerWidth;
    k = k + bytesPerWidth;
}
result.UnlockBits(bmd);
    }
```

8. STATE-OF-THE-ART

Programming for devices with limited capabilities and especially for embedded systems, has a long standing tradition of being custom work (that involves good knowledge of the hardware) that is generally not easily portable. Such devices had generally a long time-to-market and a long term market presence. With the proliferation of all kinds of consumer devices with a relatively short market presence, this is not longer feasible. Research has gone into virtual machines that provide a stable and effective environment for the application programmer across different types of devices. The abstractions provided by the virtual machine vary, and are tied to the intended model

of usage. For embedded and small systems, an important aspect is a small footprint and therefore these VMs provide less functionality than the typical desktop VM.

There is numerous research to improve garbage collection on embedded systems ([5], [6], [7]), to lower power consumption ([8], [9], [10]) and to improve code compactness ([11], [12]).

Other work focusses on a specific use pattern or environment to reduce the footprint of the virtual machine. Maté [13] is a virtual machine especially targeted for sensor networks. EarlGray [14] is a component based Java virtual machine for embedded systems. Also, there is ongoing research to have an effective and user friendly user interface across different systems [15],[16].

9. CONCLUSION

We have ported a desktop program to two popular platforms for program development on tiny devices; J2ME and the .NET Compact Framework. Both platforms are good choices for this purpose. J2ME is more configurable and provides a richer environment. Because of that J2ME may be the better choice for very limited devices, eg. smartcards. When carefully programmed, the platform independence is an advantage over .NET. We also found the documentation available for J2ME to be of higher quality. .NET Compact is a one size fits all solution that supports advanced C-like programming features and has remarkable library interoperability support. Porting a J2SE program to J2ME is much easier than porting the .NET Framework version to the .NET Compact Framework. We also identified some problems in the .NET Compact Framework. The memory footprint and memory usage of the two platforms is not decisively different for our application. For Draco, the .NET Compact Framework implementation has a performance advantage over the J2ME implementation.

Both platforms lack functionality to explicitly manage power usage. Although these platforms hide low details from the programmer, it remains very important to have an insight in the internal working of the platform to get the best performance.

10. REFERENCES

- [1] Yves Vandewoude et al. Draco: An adaptive runtime environment for components. *Technical Report CW372, CS Dep. KULeuven*, <http://www.cs.kuleuven.be/yvesv/?q=Draco>.
- [2] Roberto Costa and Erven Rohou. Comparing the size of .net applications with native code. In *CODES+ISSS '05: Proc. of the 3rd IEEE/ACM/IFIP Int Conf on H/S codesign and system synthesis*, pages 99–104. ACM Press, 2005.
- [3] ECMA. Cmu 1394 digital camera driver. <http://www-2.cs.cmu.edu/iwan/1394/index.html>.
- [4] Microsoft. Mixed dll problem. <http://support.microsoft.com/?id=814472>.
- [5] David F. Bacon et al. Garbage collection for embedded systems. In *EMSOFT '04: Proc. of the 4th ACM Int Conf on Embedded software*, pages 125–136, New York, NY, USA, 2004.
- [6] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *ISMM '02: Proc. of the 3rd Int symposium on Memory management*, pages 150–156, New York, NY, USA, 2002.
- [7] Guangyu Chen et al. Improving java virtual machine reliability for memory-constrained embedded systems. In *DAC '05: Proc. of the 42nd annual conf on Design automation*, pages 690–695, New York, NY, USA, 2005.
- [8] Paul Griffin et al. An energy efficient garbage collector for java embedded devices. In *LCTES'05: Proc. of the 2005 ACM SIGPLAN/SIGBED conf on Languages, compilers, and tools for embedded systems*, pages 230–238, New York, NY, USA, 2005.
- [9] Paul Griffin et al. On designing a low-power garbage collector for java embedded devices: a case study. In *SAC '05: Proc. of the 2005 ACM symposium on Applied computing*, pages 868–873, New York, NY, USA, 2005.
- [10] P. Stanley-Marbell and L. Iftode. Scylla: a smart virtual machine for mobile embedded systems. In *WMCSA '00: Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '00)*, page 41, Washington, DC, USA, 2000. IEEE Computer Society.
- [11] Nik Shaylor et al. A java virtual machine architecture for very small devices. In *LCTES '03: Proc. of the 2003 ACM SIGPLAN conf on Language, compiler, and tool for embedded systems*, pages 34–41, New York, NY, USA, 2003.
- [12] B. De Sutter et al. On the side-effects of code abstraction. In *LCTES '03: proc. of the 2003 ACM SIGPLAN conf on Language, compiler, and tool for embedded systems*, pages 244–253, New York, NY, USA, 2003.
- [13] Philip Levis et al. Mate: A tiny virtual machine for sensor networks. In *Int Conf on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.
- [14] Ishikawa H. et al. A component-based java virtual machine for embedded systems. In *8th IEEE Int Symposium on Object-oriented Real-time distributed Computing*, pages 403–409, 2005.
- [15] Giulo Mori et al. Design and development of multidevice user interfaces through multiple logical descriptions. In *IEEE transactions on software engineering*, volume 30, August 2004.
- [16] Peter Rigole et al. A component-based infrastructure for pervasive user In *International Workshop on Software Techniques for Embedded and Pervasive Systems STEPS'2005*, May 2005.