Porting the .NET Compact Framework to Symbian Phones – A Feasibility Assessment

Alain Gefflaut, Friedrich van Megen, Frank Siegemund, Robert Sugar

European Microsoft Innovation Center
Ritterstr. 23
D-52072 Aachen, Germany
{alaingef|fmegen|franksie|rsugar}@microsoft.com

ABSTRACT

As a result of the increasing availability and processing capacity offered by portable devices, it is important for software providers to offer mobile services that seamlessly interoperate with business applications. However, currently there is still a considerable technology gap between building .NET applications on PC-like systems and programming mobile services on mid-range portable devices, a large number of which run the Symbian operating system. As Microsoft has built its .NET Compact Framework Common Language Runtime (CLR) for high-end mobile devices, it would be desirable to bring a reasonable subset of this technology to mid-range smartphone devices as well. Such a platform for executing .NET applications on Symbian-enabled smartphones has then the potential (1) to considerably facilitate the migration of .NET applications to portable devices and (2) to increase the interoperability between software running on stationary systems and mobile services. In this paper, we present an initial feasibility assessment for porting the .NET Compact Framework to Symbian smartphones, and analyze how the unique characteristics of the Symbian operating system affect the portability of the .NET Compact Framework. Based on our experiences in porting parts of the .NET Compact Framework to Symbian, we illustrate code portability between different platforms and provide a preliminary performance analysis of the .NET Compact Framework compared to Java.

Keywords

.Net Compact Framework – Symbian – Mobile Services – Smartphones – Software Migration.

1. INTRODUCTION

During the last two decades, mobile phones have become almost ubiquitous. As a result of this development, it is increasingly important for software providers to offer mobile services that seamlessly interoperate with their business applications in order to improve customer satisfaction and service availability. The .NET Framework has been a popular platform for creating such applications and services both on stationary computers and Windows CE-based PDAs. However, a large number of today's smartphones are currently based on the Symbian operating system, for which applications are either developed in Symbian C++ or Java. According to a recent study [Gar04], 80% of all smartphones shipped in the 3rd quarter of 2004 were Symbian phones. Hence, for the next couple of years Symbian smartphones are likely to remain an important platform for implementing mobile services.

As a consequence, it would be beneficial if .NET applications could also be executed on Symbian-enabled devices. .NET developers could then reuse their code for mobile services instead of reimplementing their applications from the ground up using C++ or Java. Reimplementation can be especially cumbersome since commonly used CLR/.NET features may not be present in different programming models (e.g. floating point support is absent in some J2ME profiles, SOAP Web Services support may be missing, XML and graphics programming model might differ). These issues mean that direct code reuse is not possible, which results in

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'2005 *conference proceedings,*
ISBN +#ž+), ' &ž#SžS
Copyright UNION Agency – Science Press, Plzen, Czech Republic

increased costs and is likely to introduce new program errors. Having a Common Language Runtime (CLR) running on Symbian smartphones also implies that developers could implement applications for this platform using the same programming environment and tools offered for the .NET Framework. We would like to argue that such an approach has the potential to considerably simplify the migration of .NET applications to mobile devices and makes it easier for software developers to design mobile services that interoperate with stationary .NET applications.

In this paper, we investigate whether it is feasible to port the .NET Compact Framework to Symbian, and report on our preliminary experiences in porting parts of the .NET Compact Framework to this platform. The paper also contains an analysis of specific characteristics of Symbian and describes how the internals of the Symbian operating system affect the portability of the .NET Compact Framework. Furthermore, we provide a preliminary performance analysis of executing applications for Symbian smartphones by means of the Common Language Runtime (CLR).

The remainder of this paper is structured as follows: The following section summarizes related work. Sect. 3 provides an overview of the .NET Compact Framework architecture. Sect. 4 reports on our experiences in porting parts of the .NET Compact Framework to Symbian phones and shows how we dealt with the specific demands of the Symbian operating system. In Sect. 5 we evaluate our implementation in comparison to Java. Sect. 6 gives an outlook on future work, while Sect. 7 concludes the paper.

2. RELATED WORK

The number of programming languages targeting the Common Language Infrastructure (CLI) has been steadily increasing over the years. Besides the variety of currently supported programming languages, however, CLI run-time technologies have also become increasingly interesting for simplifying the development process across different platforms and operating systems. Examples for this development are Microsoft's Rotor and 3rd party Mono and DotGNU implementations of the CLI [Rotor,Mono,DotGNU]. The last years have therefore shown a shift from using CLI technologies for language integration on a single platform to improving the development of applications across different platforms and operating systems. As the CLI has been accepted as an international standard, the development into this direction of cross-platform interoperability of CLI languages is likely to persist.

While there are significant projects that aim at supporting .NET on operating systems such as Unix and MacOS, the major difference in hosting the CLI on the Symbian operating system is that the latter is explicitly targeting resource-restricted mobile devices. Constraints regarding the amount of available memory, computational resources, and restrictions in the functionality provided by the operating system pose therefore new demands on the portability of the .NET Framework. Because of these constraints, this paper focuses on the .NET Compact Framework [NETCF] – which itself was designed for mobile devices and first implemented to run on Windows CE. Because of this, it already considers some of the typical constraints of mobile platforms.

Most Symbian smartphones are shipped with a Java Virtual Machine (JVM) already installed on the phone (J2ME MIDP, the Java 2 Platform Micro Edition Mobile Information Device Platform targets resource-restricted mobile devices such as mobile phones). A .NET Compact Framework implementation for smartphones should therefore be at least comparable to Java implementations with respect to provided functionality and resource consumption. Besides this fact, there are however major differences between Java and .NET that make a direct comparison difficult: (1) Java byte code is often interpreted while the CLR primarily uses Just-in-Time (JIT) compilation. (2) There are international standards for the CLI and C#, while there is no such standard for Java (there is a Java Community Process, though). (3) .NET supports many programming languages – with J# also a flavour of Java. This can make direct comparison difficult because this advantage can imply architectural decisions affecting the performance of the CLI. (4) The .NET Compact Framework comes with functionality that is not natively supported by J2ME MIDP. However, there are a range of publicly available add-ons and class libraries that support much of this functionality also on this Java platform [J2MEWeb].

Rashid et al. [RTCE04] compare the performance of native Symbian code with interpreted Java applications, and Raghavan et al. [RSL04] reports on a model-based performance evaluation of applications on mobile devices. In the scope of our work, test suites provided by IBM [IBMBenchmarks], covering basic features such as method calls, thread creation, and data access, were used to carry out performance comparisons.

There are several papers (e.g., [Opera] and [Helix]) dealing with some of the obstacles that arise when porting applications to the Symbian operating system. Some of the described approaches are also applicable

in the context of our work and helped us find a direction for our project.

3. ARCHITECTURE OVERVIEW

Fig. 1 gives an overview of the .NET Compact Framework architecture and its underlying components. As can be seen, the major constituents of this general architecture are (1) the actual hardware of the mobile device, (2) the operating system that provides access to this hardware, (3) the .NET Compact Framework CLR, which maps the instructions of a (4) .NET application onto instructions for the operating system and the underlying hardware.

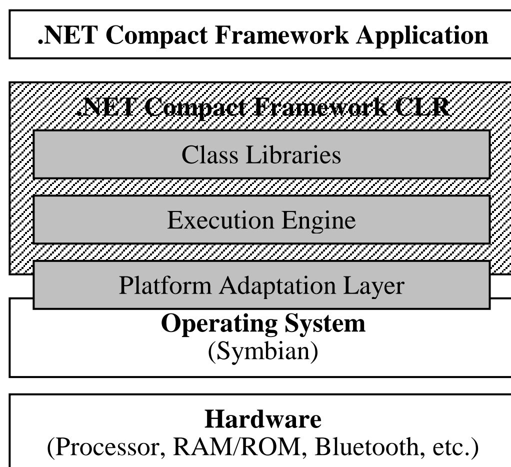

Figure 1: Overview of the .NET Compact Framework Architecture

In the following, we will shortly describe these individual components before we present our experiences in porting parts of the .NET Compact Framework to Symbian.

Hardware Constraints

A crucial aspect when trying to target a different computing platform for .NET is to be aware of the computational and functional restrictions of the underlying hardware.

The Symbian Web site currently (February 2005) lists 31 different Symbian OS phones, of which 13 are distributed by Nokia, 7 were built by Fujitsu for NTT DoCoMo's FOMA network, 3 are from Sony Ericsson, and the others come from companies such as Siemens and Motorola. For 21 of these 31 phones, for which more detailed information could be found, we looked more closely at the technical specifications.

All of the investigated phones were built around ARM processors or variants such as the OMAP 1510 from Texas Instruments, which itself is based on an ARM architecture. The processor speed varied from 104 MHz for the ARM4T processor to 220 MHz for an ARM5 CPU. As an average, most phones are operated at processing speeds of up to around 150 MHz. Regarding display capabilities, approximately 50% of the investigated Symbian smartphones have a screen resolution of 176x208 and the others a resolution of 208x320. An exception is the Nokia 9290 Communicator with a screen resolution of 640x200. This relatively large screen, however, is only used in the PDA mode of the device.

All of the smartphones we compared with each other supported Java, and most new phones come with Java MIDP 2.0 support. Furthermore, Bluetooth has become a wireless communication standard that is implemented by virtually all Symbian smartphones. In some of the new phones Bluetooth is even preferred over infrared; these phones are not equipped with an infrared port. This is important because the .NET Compact Framework provides special classes facilitating networking and communication over infrared links. In a port of the Compact Framework to Symbian-enabled devices, it therefore seems reasonable to focus more on Bluetooth than infrared as the standard interface for short-range communications.

The most striking difference when comparing Symbian smartphones is in the amount of memory integrated into the devices. While some Nokia phones such as the Nokia N-gage or the Nokia 7650 have only about 4 MB of internal memory to store photos and messages, newer models such as the Nokia 6630 come with 10 MB of memory integrated (only about 6 MB of which are free to store programs or photos); the Nokia 7710 has up to 90 MB of internal memory [MobileReview]. With respect to non-volatile memory, most phones offer the possibility to insert multimedia cards (MMC) in order to increase storage capabilities. Furthermore, the trend towards more sophisticated digital cameras integrated into smartphones will increase the demand for non-volatile memory. As a consequence, it will not be the limiting factor when porting the .NET Compact Framework to Symbian phones. A more pressing problem is the amount of RAM available on smartphones. According to [MobileReview], the amount of volatile memory available on the Nokia N-Gage, the Nokia 7610, and the new Nokia 6630 is a mere 379 kB, 1403 kB, and 8758 kB, respectively.

Tab. 1 compares typical hardware features of Symbian smartphones with those of a Compaq iPAQ PocketPC – a relatively old iPAQ model on which the .NET Compact Framework, however, successfully runs in a Windows CE based OS (newer Pocket PC's which also run the .NET Compact

Framework have significantly greater resources). As we can see, the most relevant physical difference between the iPAQ and the smartphones is the amount of memory integrated into the devices. Following an exploratory approach, we tried to assess the memory demands of a .NET Compact Framework for smartphones by porting parts of the framework to the Symbian platform (cf. Sect. 5). Considering the other hardware characteristics both platforms are somewhat similar, so that none of the hardware constraints found on smartphones should make it impossible to port the .NET Compact Framework to this platform.

Table 1: Typical hardware characteristics of Symbian smartphones compared to that of an iPAQ H3650

	iPAQ H3650	**Smartphones**
OS	Windows	Symbian
Processor	206 MHz Intel StrongARM	up to 220 MHz ARM architecture
Memory	32 MB RAM 16 MB Flash	typ. <<10 MB RAM typ. < 10 MB Flash
Display	240x320 touch screen	176x208 or 208x320 typ. no touch screen
Connect	IrDA, Bluetooth	Bluetooth, IrDA

Operating System

The second layer in our overall architecture (cf. Fig. 1) is made up of the operating system, in our case the Symbian OS. In many respects does the Symbian OS considerably differ from Windows CE, which has been the standard platform for hosting the .NET Compact Framework CLR implementation. These differences affect such elementary features as multitasking, error handling, file access, and networking. They have therefore a significant impact on our goal to port the .NET Compact Framework.

Here are some of the Symbian characteristics that so far caused most of the problems in our project (for a more detailed description of these issues, please refer to Sect. 4):

- A C++ dialect that redefines basic language structures
- No writable global and writable static variables allowed in DLLs
- Extensively used client/server model that, for example, implies constraints for accessing file and networking functions
- Event-driven programming model with a focus on non-preemptive multitasking
- Symbian's error handling and cleanup model
- Concepts from the Unix/Windows world such as environment variables as well as several file and networking functions are missing

CLR Architecture Overview

The .NET Compact Framework CLR is made up of the following main components: (1) class libraries, (2) execution engine, and (3) platform adaptation layer.

The goal of the .NET Compact Framework class libraries is to provide a basic set of classes, interfaces, and value types that constitute the foundation for developing applications in .NET. For example, support for integers, boolean values or strings, functionality for performing I/O, classes for handling exceptions, and methods for collecting information about loaded classes are all included in the class libraries of the .NET Compact Framework.

The execution engine is the core component of the CLR – it provides the fundamental services necessary for carrying out managed code. While the execution engine consists of a large number of individual components, some of its most important parts are: (1) a just-in-time (JIT) compiler (or alternatively an interpreter), (2) a garbage collector, and (3) a class and module loader. The decision whether to use a JIT compiler or to immediately carry out generated instructions in an interpreter depends on the resource constraints of a given platform. Our preliminary port is based on a JIT compiler, not an interpreter.

Because the design of the .NET Compact Framework anticipated operating system portability, access to core operating services occurs through a PAL layer. The main responsibility of the platform adaptation layer (PAL) is to map calls from the execution engine to functions provided by the underlying host operating system. In other words, the PAL serves as the main mediator between the operating system (Symbian OS in our case) and the CLR. As a result of the architectural design of the .NET Framework, the PAL is the core component that needs to be reimplemented when porting the .NET Compact Framework to Symbian OS. To illustrate the responsibility of the PAL, let us consider the example of a simple Web request. Using .NET class libraries, the code for retrieving a Web page in C# could look like this:

```
   WebRequest req;
   WebResponse resp;

4: req = WebRequest.Create(
      "http://www.microsoft.com");
5: resp = req.GetResponse();
```

Classes such as WebRequest and WebResponse belong to System.Net and are therefore part of the

class libraries provided by the .NET Compact Framework. The method calls in lines 4 and 5 of the above code result internally in a number of function calls to the underlying operating system. First, the URL "http://www.microsoft.com" must be internally resolved into a corresponding IP address. Afterwards, a timer is created with a callback function that is executed when the Web page is not retrieved in a certain time frame. Finally, a TCP socket must be created and configured that is used to send a request to and retrieve data from the remote Web server. The implementation of the class libraries in the .NET framework thereby assumes the existence of certain hooks for handling timers and dealing with sockets on the operating system layer. The PAL implements these function hooks based on the capabilities of the underlying operating system. In case of Windows CE, these mappings to function calls of the operating system are often straightforward. However, with Symbian it can be much more complicated to find appropriate mechanisms to implement the desired semantics.

4. PORTING THE .NET COMPACT FRAMEWORK

In this section, we describe our port of selected components of the .NET Compact Framework to Symbian-enabled mobile devices. Again, we would like to point out that our work focuses on evaluating whether it is feasible to port the .NET Compact Framework to Symbian phones. As a result, simple solutions were often preferred over more complex approaches in order to get a simple version of the Framework working as soon as possible.

In this section, we attempt to analyze the characteristics of the Symbian operating system that caused most of the problems in our project, and propose solutions for dealing with these issues.

Current Status

The preliminary port presented in this paper is based on the Microsoft .NET Compact Framework implementation version 1 for Windows CE. Currently, it is possible to execute basic console-based .NET applications on two Series 60 phones that are based on the Symbian OS: Phone A (OS v6.1 , 3 MB available memory, and a 104 MHz processor) and Phone B (OS v8.0a, 10MB of available memory, and a 220 MHz processor). Furthermore, we support file access and simple networking. To achieve that, work has not only been done on several Platform Adaptation Layer (PAL) modules such as threading, event handling, console output, file access, and networking, but also on the surrounding components that are used to load .NET DLLs and to start .NET applications.

C++ Dialect

The flavor of C++ used to implement native Symbian applications caused several problems in our project. In particular, Symbian C++ introduces some peculiar language features and programming models that were partly introduced because of the limited device capabilities of Symbian smartphones and partly due to historical reasons [Nok04]. Important issues are: (1) different standard data types, (2) a missing libc, (3) a special exception handling mechanism, and (4) a different memory management model.

First, simple types such as `int` or `unsigned long` are not recommended by the Symbian Software Development Kit (SDK), so types such as `TInt` and `TUInt32` had to be used instead. The STL (Standard Template Library) is also not supported due to size limitations.

Second, as a `libc` is not supported by Symbian, a basic implementation had to be attached to our project containing memory management (like `memcmp`) or C-type string manipulation functions (such as `strlen`).

Third, the GNU C++ implementation of exception handling was not mature enough at the design time of EPOC (the old name of Symbian), thus the designers employed a more lightweight approach to error handling – the "trap harness" mechanism. A function called `User::Leave()` corresponds to the `throw` directive, while the `TRAP` and `TRAPD` macros are called instead of `catch`. Exception objects were also replaced by simple error codes.

Furthermore, as mobile phones are switched on for long periods of time, the ability to reclaim unused heap cells was crucial during the design of Symbian. Therefore, a mechanism called "two-phase-construction" is used during object creation, and a "cleanup stack" structure makes sure that every object created on the heap is destroyed after it has been used.

Writable Global and Writable Static Variables in DLLs

The Symbian operating system was built with memory-constraint devices in mind. Therefore, it tries to avoid all unnecessary allocations or wastage of main memory. To prevent allocation of memory for writable static data in DLLs, which would have to be allocated for each application, and to enable eXecution In Place (XIP), DLLs that are stored in ROM are not copied to RAM. As a consequence, the programming environment does not support writable static or global data because the segment containing these values in the DLL is not writable.

If this requirement is not a major issue when writing new applications, it becomes a major problem when

porting applications that have been designed to run on operating systems supporting writable static data. This is the case for the original Microsoft .NET Compact Framework, which usually runs on top of Windows class operating systems. Two strategies can be envisaged to solve this problem. First, rewriting the libraries was ruled out as a viable solution since the number of writable static data was too large to enable a manual rewrite of the libraries. The second strategy, which is the one we followed as a way to get a test version of the .NET Framework working as soon as possible, consists in loading in RAM all DLLs used by the .NET Compact Framework application. To reach this goal, we designed and wrote a specific loader. Starting the Framework is then realized by calling the loader. The loader is in charge of downloading in RAM the image of the .NET Compact Framework binary, as well as all libraries that it needs (including the writable data section). The loader also performs the necessary relocation in order to prepare the execution. Once relocation is done, the loader identifies the entry point defined in the .NET Compact Framework binary and jumps to its location. Although this solution works, it is far from optimal since it can result in a possibly high memory footprint. While this is not a problem in our feasibility assessment, this issue would have to be addressed in a real, complete port of the .NET Compact Framework to Symbian.

Starting .NET Applications

When a .NET application – which is usually generated using a development environment and a compiler on a Windows-based PC system – is to be executed on a Symbian phone, it must be assigned to our .NET Compact Framework implementation for execution. As .NET compilers generate files in the standard .NET portable executable file format, it is possible to distinguish any .NET application from native Symbian applications. Luckily, the Symbian OS provides the concept of so called Recognizers, which are used to assign certain file types to selected applications. For example, HTML files can be associated with a Web browser, PDF files with an Acrobat reader, etc. As this association can be based on more that just the file extension and allows us to analyze the file to be executed, we use a special Recognizer for starting .NET applications.

Dealing with Symbian's Client/Server Framework

The Symbian OS introduces a range of servers to deal with system resources on behalf of different clients. Examples for such servers are the file server, the socket server, and the window server; servers are usually located in a different process than the clients that want to access their services. The problem with Symbian's client/server framework from the perspective of the .NET Compact Framework is that only the client thread that creates resources for interacting with a server can use and destroy them. This has some implications for a port of the .NET Framework, and especially the Platform Adaptation Layer (PAL). Imagine that there is a .NET application consisting of two threads that both want to access a file. In this scenario, the PAL would be responsible for mapping the file access to corresponding operating system functions. For example, there would be a function like `PALFile_Open()` that sends a request to the Symbian file server to open a file. However, since both .NET threads – which are both mapped to Symbian threads in our implementation – might want to open a file, this is not possible because only the client thread that created the connection to the file server can do that. To solve this problem, we introduced a mediator thread that handles all communication with the file server. Symbian OS threads that represent threads in .NET then interact with this additional thread in order to access files. For the PAL implementation, this means that `PALFile_Open()` does not interact with the file server directly, but instead issues a request to the intermediary thread communicating with the file server. A similar mechanism is deployed to handle networking and console access.

Dealing with Symbian's Focus on Cooperative Multitasking

In the desktop domain, pre-emptive multitasking replaced cooperative multitasking years ago when resources became cheaper and PC-like systems much more computationally powerful. Furthermore, using pre-emptive multitasking for different computations that need to be carried out concurrently is much easier from a programmer's point of view than having to deal with the burden to split a long-running task into subtasks in order to keep up responsiveness. However, although the Symbian operating system supports pre-emptive multitasking, switching between different pre-emptive threads is considered very expensive and programmers are strongly encouraged to use cooperative multitasking instead [Nok04, Har03]. To support programmers in handling cooperative multitasking, Symbian introduced the concept of Active Objects as a programming paradigm. Together with a so-called Active Scheduler, Active Objects are supposed to facilitate the programming of non-preemptive concurrent tasks.

However, cooperative multitasking using Active Objects has still the disadvantage that if there is a long-running calculation, it only will give control to another task if it is finished. As this might severely

reduce the responsiveness of a user interface, for example, books on Symbian programming [Har03, Nok04] strongly suggests manually splitting long-running tasks into smaller subtasks that can faster pass on control to other subtasks, thereby improving the overall responsiveness of the system. This, however, does not map well with the notion of threads in .NET because threads in .NET are generally viewed as being preemptively scheduled. To deal with this issue in a port of the .NET Compact Framework there are several theoretical solutions:

(1) If there is a thread in .NET, it is possible to generate a pre-emptively scheduled thread in the Symbian operating system and accept the effect on system performance this does imply. (2) When the execution engine requests a new thread to be created for a thread in a .NET application, a new Active Object could be created that handles the associated task. However, this would mean that we would need a mechanism to automatically find a location in the code where this active object can pass on control to a different task. Finding a place where this can be done requires at least the help from the JIT compiler or special statements in the .NET code that would have to be used by a programmer. (3) Another important issue with threads is that Symbian's client/server model (see previous subsection) forces us to introduce preemptively scheduled threads on the operating system layer to sequentialize access to servers (the file server, for example). In order to reduce the number of low-level Symbian threads, it is possible to use a single thread for all different servers. The downside of this, however, is that a .NET thread that wants to output a string on the console might need to wait for a different .NET thread that wants to do file access. Whether this can be accepted depends mainly on the concrete .NET application. In the current state of our port, .NET threads are directly mapped to pre-emptively scheduled threads on the Symbian operating system layer.

5. EVALUATION

The purpose of this section is to estimate the performance of a .NET Compact Framework implementation for Symbian smartphones in comparison to other runtime environments where intermediate code is executed by a just-in-time compiler or an interpreter. To achieve this goal, we compare the time necessary to execute .NET code on our platform with the time needed to execute Java code on a Symbian smartphone. As it would be too complex to compare and difficult to interpret the runtime characteristics of complete applications written for .NET and Java – due to the different algorithms and optimizations Java and .NET runtimes might use – our approach is instead based on micro-benchmarking. Micro-benchmarks are simple programs (usually loops) targeting a single functionality such as memory allocation or thread synchronization. Because of the simplicity of the underlying programs, porting the benchmarks to both Java MIDP and .NET is relatively simple. This also assures that a comparison based on these benchmarks stays fair.

In order to carry out the evaluation, we chose a suite of micro-benchmarks originally written by IBM to measure the performance of simple Java operations in a standard Java Virtual Machine (JVM) environment [IBMBenchmarks]. These benchmarks originally targeted the desktop versions of Java and thus are using APIs that are not available on a Symbian smartphone. Therefore, we selected relevant tests from this benchmarking suite and adapted them such that they could be executed by the JVMs installed on our Symbian smartphones. As a result, benchmarks for the reflection interface of Java were omitted as well as tests targeting file access functions (file access is not supported on the smartphone JVMs used in our tests). Additionally, we also had to drop any benchmark using Java functionality not available to .NET applications.

The other major change in the benchmarks dealt with timing issues. Instead of dynamically calculating the number of iterations of a test, we hard-coded the number of iterations for each benchmark based on the duration of a test. This was done because it simplifies porting of the test framework to C#, and because it ensures that all tests are carried out the same amount of times on different devices. In general, faster tests run more often than more time-consuming tests. For the above reasons, test results measured with the selected benchmark suite on another hardware platform cannot be directly compared to the results presented in this paper.

Porting the Benchmarks to C#

In a second step, we ported the selected set of micro-benchmarks to the .NET Compact Framework using C#. Because Java is quite similar to C#, porting the micro-benchmarks required mainly small syntactic modifications. For example, the C# language keeps a different set of reserved identifiers, thus, variables named `internal` or `object` had to be renamed. Besides syntactic modifications, a few discrepancies between Java and C# forced us to modify the code.

Unlike Java, for example, C# does not support the `synchronized` tag for methods or classes. For tests that required synchronized method calls, we removed the synchronized tag and added a *`lock(this)`* as the first statement of the method. The `lock` statement in C# is used to acquire the monitor associated to an instance of a class, thereby

preventing anybody else from calling a method of this object. As a result, this statement emulates the behavior of the `synchronized` tag of Java.

Another, slightly more complex modification in the benchmarks was necessary because there is no simple alternative to the `Thread.Join()` statement in the .NET Compact Framework. This is a difference w.r.t. the original .NET Framework, but in the Compact Framework, it is difficult to ask a thread to wait for the completion of another thread. To handle this problem, we rewrote the original tests such that explicitly generated events were used for signaling.

Micro-Benchmarks Description

The first micro-benchmark in our evaluation (cf. Tab. 2) measures memory read latency by reading the elements of an array. The second micro-benchmark measures the efficiency of calling a single method. The test distinguishes between calling a plain and a synchronized method. The third micro-benchmark deals with thread creation. This test sequentially creates threads and waits for them to start. Since the Symbian documentation in many places warns against the overhead involved when creating threads we were especially curious how well our implementation behaves compared to the Java thread implementation. The fourth micro-benchmark measures the time necessary to create new objects and the overhead caused by inheritance. In particular, it tests the creation of small objects derived over two generations compared to the creation of large objects that also inherit from a baseclass over two generations. This test also illustrates the performance of the memory subsystem and to some extend of the garbage collector. The fifth micro-benchmark measures the performance of comparing strings. The last three tests concentrate on measuring the performance of general array handling operations (e.g., initialization and copying). Both Java and C# provide support for a system-level array copy function a programmer should use for performance reasons. The `CopyArray` test therefore has two versions, one using the system-level function, the other using a naive copy of the array using a loop. While this might result in a performance penalty for a runtime that interprets code, we do not expect a big performance hit when code is generated by a JIT compiler. Similarly, the `InitArray` and `SumArray` micro-benchmarks provide two versions, one using a simple loop, the other using unrolling to limit the cost of the loop overhead.

Results Analysis

Tab. 2 shows the results obtained by executing the described tests on different platforms and execution environments. For the analysis of the results, the reader should keep in mind that the .NET tests for Symbian smartphones were carried out on a preliminary port of the .NET Compact Framework.

The first column of Tab. 2 shows the name of the micro-benchmark. The second lists the parameters used to run the micro-benchmark (starting with the number of iterations). Columns three and four show the results, in milliseconds, of the Java micro-benchmarks when executing them on the JVMs that were already installed on the smartphones used for our experiments (cf. Sect. 4). The next three columns show the results when carrying out the benchmarks in a .NET Compact Framework runtime. As can be seen in the table, we have used our port on Phone A and Phone B (cf Sect. 4) for the tests and compared these results with a standard .NET Compact Framework running on a regular PDA (a T-Mobile MDA II running PocketPC 2003 has been used for this experiment). Although not directly comparable, the results obtained with the PDA are useful to find out whether performance differences between Java and .NET are a problem of our PAL implementation or shared between .NET runtimes on different platforms.

As a general result, the speed of our initial port of the .NET Compact Framework is comparable with the corresponding Java implementation on Phone B and sometimes significantly faster on Phone A. A likely reason for this is that the JVM on Phone A seems to use an interpreter, while Phone B comes with a JIT. In two occasions, however, our port of the .NET Compact Framework is much slower than the Java runtime on the same device. These correspond to tests calling synchronizes methods (we are 4.8 times slower on Phone A) and spawning threads (we are 52 times slower on Phone A).

In case of synchronized methods, the Java implementation of a synchronized method call takes twice as long as calling a method that is not synchronized. It is remarkable, however, that this is much faster than the time needed in our port, where calling locked method is 157 times slower than an unsynchronized method call on Phone A. We expected calls to a synchronized method to be slightly slower compared to the unsynchronized version. Furthermore, since there is no real concurrency involved (as only one thread in this test calls the functions), we did not expect a major difference. Our first assumption was that our implementation of the corresponding PAL functions were responsible for the poor performance.

Table 2: Time for running benchmarks (in ms)

Test	Parameter	Java		.NET Compact Framework		
		Phone A	Phone B	Phone A	Phone B	PDA
1. MemReadLatency	#1000000, 4, 512	1578	141	219	110	122
	#1000000, 8, 256	1547	125	219	109	121
2. Method Calling	#1000000, internal, sync	4094	579	19703	32390	12843
	#1000000, internal, nosync	2719	203	125	62	330
	#1000000, external, nosync	2703	219	172	79	394
3. Spawn Threads	#1000, <>	422	1437	21937	15062	2579
4. AllObjectConstruct	#10000, small, assign, 3	219	31	63	94	61
	#10000, large, assign, 3	1125	250	ENOMEM	219	103
5. StringCompare	#10000, 128	2500	328	531	250	217
	#10000, 512	9187	1157	2047	984	854
6. CopyArray	#10000, 1024, simple	3890	328	250	375	389
	#10000, 1024, system	203	250	531	687	69
7. InitArray	#10000, 1024, unrolled	1547	250	31	234	166
	#10000, 1024, simple	3438	235	16	250	271
8. SumArray	#1000, 512,simple	187	16	531	0	15
	#1000, 512,unrolled	94	16	2047	0	12

Comparing this to the tests running on the PDA, however, revealed that the real reason might partially reside in the implementation of the Compact Framework itself. This is because even on the PDA locked code runs 39 times slower than a function not using the `lock` statement (cf. previous subsection). Spawning a thread is also considerably slower in our Symbian .NET Compact Framework implementation than in the Java implementation.

Right now, we are not sure if this is due to a bad implementation in our PAL layer or to the use of different synchronization primitives in our adaptations of the micro-benchmarks. The result for the same test on the PDA seems to indicate that it is a problem of our implementation on Symbian, and we are currently in the process of identifying the underlying problem.

As can be seen in Tab. 2, one of the tests (`AllObjectConstruct` with large objects) failed with an out-of-memory error on Phone A. A possible explanation for this problem is that the garbage collector was not able to reclaim memory as quickly as the test requested new objects to be created. To confirm this theory, we modified the test to manually call the garbage collector during the test. This solved the problem, but did not allow us to report useful results since the reported time to execute the benchmark included the time to run the garbage collector. Solving this issue is a work item for us that we will investigate in the scope of our project.

6. FUTURE WORK

Security

So far, we have not explicitly dealt with security in our project, but there are a number of security features that could be addressed in the future. These features could be divided into managed code security and the .NET Framework security.

Managed code security generally follows the guidelines of the .NET Compact Framework, which currently allows full access to resources through the P/Invoke mechanism (which allows for calling functions of the underlying OS). Later releases of the .NET Compact Framework will support security policies, custom permission sets, imperative and declarative security checks [MSDNSecurity].

Our .NET Compact Framework runtime itself is a Symbian application, thus special attention needs to be placed on testing the implementation against possible exploits – especially the PAL layer, which has access to core OS features.

Porting the GUI

Symbian allows access to the GUI on several layers. The OS itself provides a common graphics server that provides the main window, basic drawing functions, and event handling mechanisms. Direct screen access is also possible. On top of that there are several phone-specific graphic libraries, the most common being the AVKON library built for Series 60 phones.

Three distinct approaches were identified that could be followed in implementing the GUI:

1. Using basic drawing primitives to adapt an existing portable graphics library to Symbian smartphones. This approach would be the easiest to implement, but it would probably result in a high memory footprint and a slow performance of the UI subsystem. The look-and-feel would also be different from native Symbian applications.
2. Mapping .NET user interface calls to AVKON. This would be the most convenient solution, but there are significant differences between the two APIs. Major problems include the creation of resource files that the Symbian GUI framework relies on and several threading issues that prevent multiple threads to access the same control or have a parent-child window relationship.
3. Providing access for the AVKON GUI: This would place the burden of dealing with a device-specific library on the .NET developer, but proxy objects and helper functions could assist her during the process.

7. CONCLUSION

This paper evaluated the feasibility of porting the .NET Compact Framework to Symbian smartphones. Our analysis shows that the specifics of the Symbian OS and the resource constraints of today's smartphones make porting difficult but not impossible. Carrying out a serious port of the .NET framework, however, would require a considerable amount of manpower in order to appropriately react to the constraints of the Symbian platform. Our comparison with Java showed that .NET programs executed on smartphones would have similar performance characteristics. This is a very promising result and speaks in favour of the overall design of the .NET Compact Framework for resource-constraint devices.

8. ACKNOWLEDGMENTS

We would like to thank Gerd Rausch for helping us with the implementation. We thank Wolfgang Manousek, Mark Gilbert, and Ivo Salmre for all the competent comments regarding the .NET Compact Framework and Symbian, and their continuous support throughout this project.

REFERENCES

[DotGNU] The DotGNU project, http://www.dotgnu.org.

[Gar04] Gartner. Market Share: Smartphones, Worldwide, 3Q04, http://www3.gartner.com/DisplayDocument?doc_cd=125555, December 2004.

[Har03] Harrison, R. Symbian OS C++ for Mobile Phones, Wiley & Sons, August 2003.

[Helix] Wright, G. The Symbian porting project on HelixCommunity.org, https://symbian.helixcommunity.org/.

[IBMBenchmarks] The jMocha Microbenchmark Framework and Suite for Java, http://www-124.ibm.com/developerworks/oss/jmocha/index.html.

[J2MEWeb] Sun Microsystems: Java 2 Platform, Micro Edition (J2ME) Web Services, Technical White Paper, July 2004.

[MobileReview] Mobile Review Web site, http://www.mobile-review.com.

[Mono] The Mono project, http://www.mono-project.com/.

[MSDNSecurity] Microsoft Developers Network: Security Goals for the .NET Compact Framework,http://msdn.microsoft.com/library/en-us/dv_evtuv/html/etconSecurityGoalsForNETCompactFramework.asp.

[NETCF] The Microsoft .NET Compact Framework, http://msdn.microsoft.com/smartclient/understanding/netcf/.

[Nok04] Edwards, L.; Barker, R. Developing Series 60 Applications, A Guide for Symbian OS C++ Developers, Addison-Wesley, 2004.

[Opera] Porting Opera to EPOC, http://www.symbian.com/developer/techlib/papers/khopera/opera%5Fkeithhollis.htm.

[Rotor] Stutz, D. The Microsoft Shared Source CLI Implementation, Microsoft Corporation, Online MSDN article, http://msdn.microsoft.com/library/en-us/dndotnet/html/mssharsourcecli.asp, March 2002.

[RSL04] Raghavan, G.; Salomaki A.; Lencevicius, R. Model Based Estimation and Verification of Mobile Device Performance, Fourth ACM International Conference on Embedded Software (EMSOFT '04), Pisa, Italy, pp. 34-43, September 2004.

[RTCE04] Rashid, O.; Thompson, R.; Coulton, P.; Edwards, R. A Comparative Study of Mobile Application Development in Symbian and J2ME using Example of a Live Football Results Service Operating over GPRS. IEEE International Symposium on Consumer Electronics, Reading, UK, pp. 203-207, September 2004.