JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 3, Special issue: .NET Technologies 2005 Conference, 2006

Porting the .NET Compact Framework to Symbian Phones

Frank Siegemund, Robert Sugar, Alain Gefflaut and Friedrich van Megen, European Microsoft Innovation Center

Abstract

This paper presents our experiences in porting selected parts of the .NET Compact Framework to Symbian smartphones. Our port includes support for basic services such as threading and file access, low-level networking modules as well as Web Services. We also present a portable .NET GUI for the Symbian platform. The paper shows how the programming models of .NET can be efficiently mapped to the runtime structures provided by operating systems for resource-constrained devices such as the Symbian OS. In a detailed analysis, we compare the performance of our port to that of Java and native code.

1 INTRODUCTION

During the last two decades, mobile phones have become almost ubiquitous. As a result of this development, it is increasingly important for companies to offer applications for mobile users that seamlessly interoperate with server-based business software 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 mobile devices. While Windows CE-based smartphones are getting increasing attention, however, a number of phones are currently based on the Symbian operating system. According to a recent study by Canalys [Canalys], the Symbian OS owned 63.2 % of the mobile device OS market at the end of the third quarter 2005.

A core problem of the Symbian OS is that application development on this platform is considered to be difficult. Taking into account the need for companies to offer services to a broad range of mobile users, it would therefore clearly be advantageous to have a .NET Common Language Runtime (CLR) available on Symbian-enabled devices. .NET developers could then reuse code instead of reimplementing applications from the ground up in an entirely different programming environment. Reimplementation can be especially cumbersome since commonly used CLR/.NET features may not be natively present on different programming platforms [Sin03] (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 increased costs and is likely to introduce new program errors. Having a CLR running on Symbian smartphones implies that developers could implement applications for this platform using the same programming environment and tools offered by the .NET Framework. With respect to convenience, this would be a significant step forward for application development under Symbian.

In this paper, we present our experiences in porting selected parts of the .NET Compact Framework to the Symbian operating system, and report on the major design decisions that had to be made during this work. The main challenge was to map the programming model of the .NET Compact Framework (or, more precisely, of .NET programming languages like C#) onto the programming model provided by the Symbian OS. Because of the considerable gap between these two models, which is a result of the resource constraints of many Symbian-enabled devices, porting .NET technologies to Symbian poses specific problems that are addressed in the remainder of this paper. After having discussed these issues, the paper gives an overview of the memory requirements of our port and provides a detailed performance analysis of basic services such as string handling and of our GUI implementation.

In summary, the paper answers the following core questions:

- Is it feasible to have a .NET Compact Framework CLR running on resource-constrained Symbian smartphones?
- What are the main obstacles when porting the .NET Compact Framework to Symbian?
- Is it possible to implement a portable .NET GUI on resource-constrained Symbian smartphones?
- What is the expected performance of carrying out .NET applications on the targeted device platform?

The remainder of this paper is structured as follows: The following section summarizes related work. Sect. 3 presents the current status of our port. Sect. 4 shortly reviews the .NET Compact Framework architecture, provides information about the Symbian operating system, and compares the hardware constraints of Symbian smartphones with those of other .NET hardware platforms. Sect. 5 reports on our experiences and the major design decisions we had to make while porting the .NET Compact Framework to Symbian phones. It also shows how we dealt with the specific characteristics of Symbian and its programming model. In Sect. 6 we evaluate our implementation, and Sect. 7 concludes this paper.

2 RELATED WORK

The number of programming languages targeting the Common Language Infrastructure (CLI) [CLI,MG00] has been steadily increasing over the last years. Besides the variety of available programming languages [BKR04,Gut01, Ham03,SSM00], however, CLI-compliant virtual execution systems are also increasingly used to facilitate application development across different hardware platforms and operating systems. Examples for this development are Microsoft's Rotor and 3rd party Mono and DotGNU implementations of the CLI [DotGNU,Mono, SNS03]. While these implementations aim at supporting .NET on operating systems such as Unix and MacOS, the major difference to our work is that we investigate an operating system that is explicitly designed for resource-restricted mobile devices. As a result, constraints with respect to the amount of available memory, computational resources, and restrictions in the functionality provided by the operating system pose challenging new problems and research questions that we needed to address.

There are papers (e.g., [Opera] and [Helix]) dealing with several 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. The migration to other hardware platforms or operating systems is recognized as one of the most difficult and error-prone processes during the lifetime of a software product [BLWG99]. George and Wong [GW04], for example, address the problems that arise when porting a powerful real-time operating system such as Windows CE .NET to a different hardware platform, while Kontogiannis et al. [KMW+98] try to automate tasks of the migration process. As for the .NET Framework, its architectural design aims at simplifying migration to a different hardware platform by means of the Platform Adaptation Layer [SNS03].

Because of the resource constraints of many Symbian smartphones, this paper focuses on the .NET Compact Framework [NetCF] – which itself was designed for mobile devices and first implemented to run on Windows CE. The .NET Compact Framework already considers some of the typical constraints of mobile platforms but does not deal with the unique constraints and programming models of operating systems such as Symbian.

Most Symbian smartphones ship with a Java Virtual Machine (JVM) already installed on the phone [RV01] (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 ME implementations with respect to provided functionality and resource consumption [Sin03]. Nevertheless, there are major differences between Java and .NET that make a direct comparison difficult: (1) Java bytecode 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 flavor of Java. This can make direct comparison difficult because the advantage of language integration 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 the Java platform.

The company AppForge [Forge] has a product called Crossfire that aims at enabling cross-platform application development on mobile devices using .NET programming languages. To the best of our knowledge, .NET applications are transformed into a device-specific custom format that is then executed on a phone. Thus, the main difference to our work is that we aim to have a complete CLI-compliant virtual execution system running directly on Symbian-enabled mobile phones. We also strive for a greater integration into the .NET programming environment by exposing the same interfaces for Web Services and GUI programming as with the .NET Compact Framework.

Building efficient graphical user interfaces on top of the Symbian OS poses challenging problems because of the resource constraints of the targeted device platform. Compared to stationary PCs and PDAs, Symbian smartphones have relatively small displays, often no touch screens, and relatively weak processors (cf. Sect. 4). Hence, several design decisions have to be evaluated when realizing GUI functionality on such a hardware platform. An interesting approach to GUI programming in the context of .NET is outlined by Bishop and Horspool [BH04], who propose an XML-based GUI description notation that allows programmers to abstract from much of the low-level issues of GUI programming. Their Views GUI engine [BW05] provides an implementation of this approach for different operating systems and adds the functionality of the System.Windows.Forms library to Rotor. Some of these concepts are used and extended in Mirrors [Mil05].

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 [jMocha] covering basic features such as method calls, thread creation, and data access were used to carry out performance comparisons.

3 CURRENT STATUS

Overview

In the current version of our prototype it is possible to execute .NET Compact Framework applications on two selected Series 60 smartphones. The Series 60 platform from Nokia was our first target because it is the most popular User Interface (UI) platform for Symbian phones at the moment. However, we would like to emphasize that our design does not rely on any UI platform on top of the core Symbian OS, but instead depends only on low-level runtime structures that are provided by the core operating system. Consequently, only a very small subset of our project – the Symbian application that starts the actual .NET execution engine – has to be rewritten when porting our work to a different Symbian UI platform. Ease of portability across different Symbian devices is important and was a major factor influencing our design decisions because there are not only many different flavors of the Symbian OS itself but also a range of different UI platforms on top of it (such as Series 60 or UIQ).

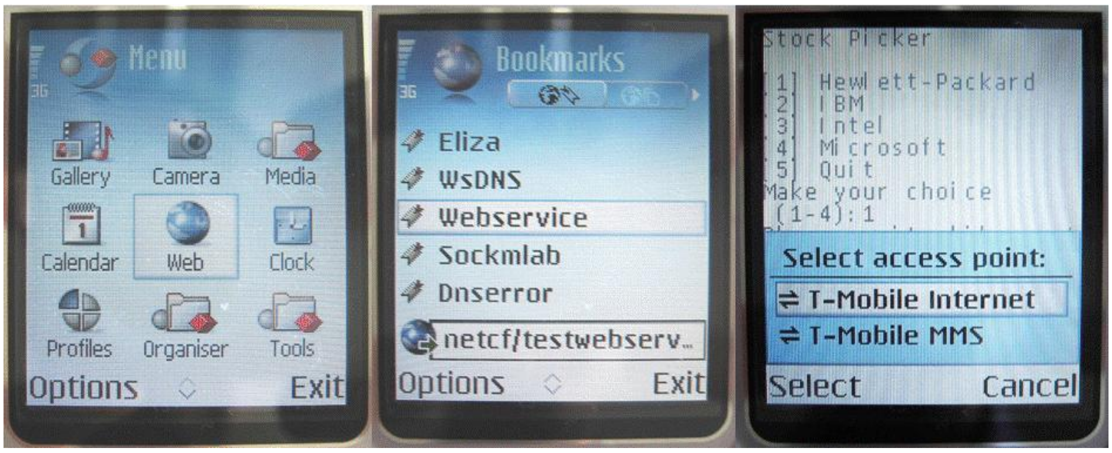

Figure 1: Starting .NET applications on a Symbian smartphone.

The core modules of functionality supported by the .NET Compact Framework CLR can be classified as (1) basic services, (2) storage, (3) networking, (4) GUI, and (5) device support [SNS03]. Our prototype for Symbian implements all main basic services such as threads, synchronization mechanisms, timers, events and mathematical support functionality. Regarding storage, it is possible to work with files and XML; XML can be read from files into internal data structures, modified in memory, as well as written back to and exchanged over Symbian's file system. The current implementation does also support low-level networking over the socket interface, Web requests, and Web Services. Of central importance in our work was to keep the existing .NET programming interfaces unchanged in order to make the development process on the Symbian platform as easy as possible. Another result of keeping the programming interfaces intact is that all the development tools available for the .NET Compact Framework are still supported. For example, programmers can make use of the established way to implement Web Services in the .NET Framework, which facilitates Web application development. Keeping programming interfaces unchanged is also important for the GUI implementation in our prototype because it implies that GUI layout tools such as the Form Designer embedded into Visual Studio can be used to design user interfaces. Because of this, our port includes a portable GUI for Symbian-enabled devices. As the last module of functionality, the .NET Compact Framework supports the interaction with various kinds of devices over IrDA, Bluetooth, or USB. Our port, however, does not support these different communication interfaces so far. Instead, Web Services operate on top of GPRS or GSM; simple messaging functionality for sending SMS has also been implemented.

Application Development and Application Startup

Considering the wide range of supported functionality, an interesting question is how a programmer can actually use this functionality to develop applications for Symbian smartphones. The simple answer to this question is that application development for Symbian smartphones does not differ from application development for Windows-based devices. In fact, the same programming environment and tools can be used during the development process. Typically, an application that has been written in a language supported by the .NET Compact Framework (e.g., C#) is compiled on a desktop computer using the standard compiler for this language (e.g., csc for C#). The result of this compilation is an EXE file in the Portable Executable (PE) file format. This file – without any modifications – is copied to a Symbian smartphone and executed there inside the .NET virtual execution system.

There are several ways to start .NET applications on a Symbian smartphone. For example, each .NET application can be exposed as a separate icon on the phone's main screen. The disadvantage of this is that an additional installation step is required when deploying the application to create the icon. Furthermore, as the phone screen itself is quite small, too many icons can reduce usability and make navigation difficult. Another possible deployment path is to start applications from within a separate tool. Users would then use this tool to browse the contents of their phone and start an application by just clicking on it. Our choice was to follow a similar approach by starting .NET applications from within the Web browser on Symbian smartphones. As can be seen in Fig. 1, a user opens the Web browser from the main phone menu. Each .NET application is then exposed as a bookmark that references a local application file. By clicking on the bookmark, the .NET execution engine is started as an embedded application inside the Web browser and automatically loads and runs the referenced application file. Fig. 1 depicts the process of starting .NET applications on Symbian smartphones based on the example of a simple text-based Web Service application.

GUI

Besides text-based applications, our prototype also supports applications with a more sophisticated user interface. As can be seen in Fig. 2, the corresponding GUI implementation is based on custom controls that are similar to that on Windows-based smartphones. The reasons for this design decision are described in more detail in Sect. 5. Currently, a range of different GUI controls are available. We support windows and dialogs, labels, text boxes, push buttons, combo boxes, radio buttons, picture boxes, and menus (see Fig. 2).

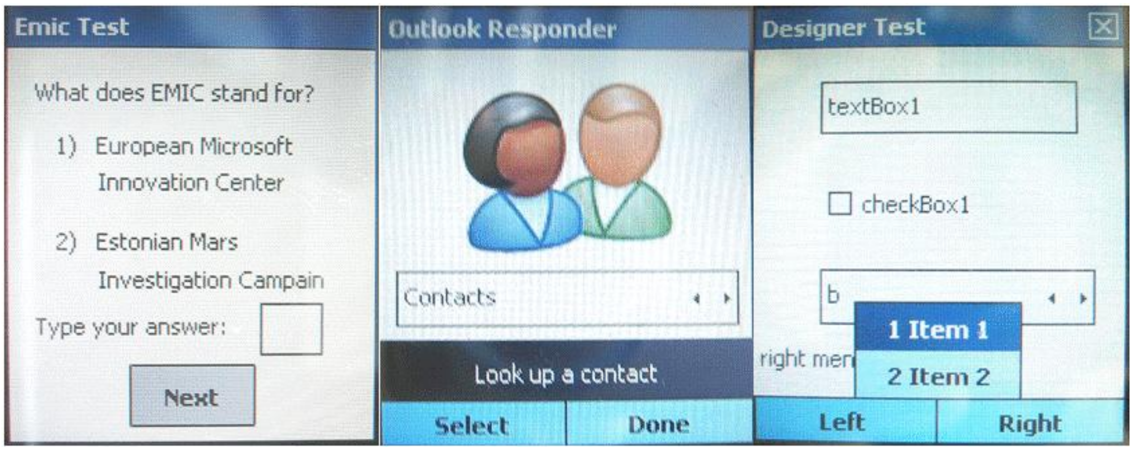

Figure 2: A portable .NET GUI on a Symbian smartphone.

4 ARCHITECTURE OVERVIEW

Fig. 3 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 runtime environment, which maps the instructions of a (4) .NET application onto instructions of the operating system and the underlying hardware [SNS03,MG00].

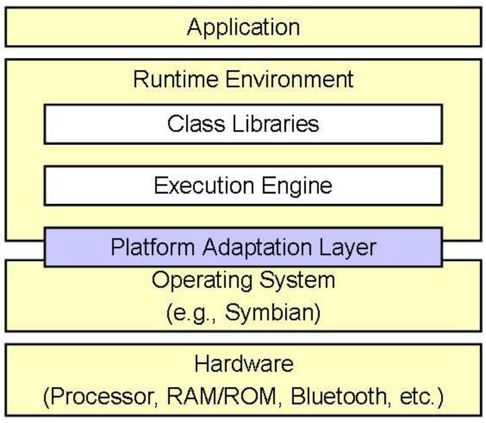

Figure 3: Overview of the .NET Compact Framework Architecture.

In the following, we will shortly describe these individual components before we present our experience 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.

In January 2006, the Symbian Web site listed 44 different Symbian OS phones, of which 19 were distributed by Nokia, 16 were built for NTT DoCoMo's FOMA network (10 from Fujitsu, 4 from Mitsubishi, 1 from Motorola, and 1 from Sharp), and the others were manufactured by companies such as Sony Ericsson or Panasonic. For 24 of these 44 phones, for which more detailed information could be found, we looked more closely at the technical specifications.

All of the investigated phones are 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 varies from 104 MHz for the ARM4T processor to 220 MHz for an ARM5 CPU. The fact that ARM processors are used in Symbian smartphones has made our port significantly easier since JIT compilers were already available for this hardware platform – most Windows CE devices also feature ARM-based cores.

The most striking difference when comparing Symbian smartphones to Windows CE devices is in the amount of random access memory available. According to [MR], the amount of volatile memory available for applications on the Nokia 3650, the Nokia 6630, and the new Nokia N90 is 1.9 MB, 7 MB, and 21 MB, respectively. This is the amount of free RAM after the OS has started. Early Symbian phones generally had less than 4 MB of RAM and only the newest models feature more than 10 MB; Windows Mobile devices have generally much more RAM than that. We suspect that the main reason to limit volatile memory apart from cost is battery consumption. Continuous refreshing of the memory modules can reduce standby time significantly. Hence, non-volatile memory (most of the new phones come with an MMC card) is preferred for storage.

	Nokia 3650	**Nokia 6630**	**Nokia N90**	**iPAQ H3650**	**Dell Axim X51v**
OS	Symbian OS 6.1	Symbian OS 8.0a	Symbian OS 8.1	Windows Mobile	Windows Mobile
Processor	104 MHz	220 MHz	220 MHz	206 MHz	624 MHz
RAM	1.9 MB free of 4 MB	7 MB free of 10 MB	21 MB free of 48 MB	32 MB	64 MB
Display	176x208	176x208	352x416 +128x128	240x320 touch	640x480 touch
Wireless	IrDA Bluetooth GPRS	Bluetooth GPRS UMTS	IrDA Bluetooth GPRS UMTS	IrDA Bluetooth	IrDA Bluetooth WLAN

Table 1: Hardware characteristics of Symbian Series 60 and Windows Mobile devices

Tab. 1 depicts typical hardware characteristics of Symbian Series 60 devices on the low-end (Nokia 3650), medium-tier (Nokia 6630) and high-end (Nokia N90, a recently introduced smartphone). For comparison the table also contains specifications of Windows CE-based devices. The iPAQ H3650 is one of the first iPAQs that featured a .NET Compact Framework, while the Dell Axim is a new model. It should be noted that other Symbian devices exist with different characteristics, such as the Ericsson P910i with a touch screen or the more powerful Series 90-based Nokia Communicators.

As a summary, we have found that none of the hardware constraints of Symbian smartphones should make it impossible or infeasible to port the .NET Compact Framework to the Symbian platform. Especially the memory constraints of Symbian smartphones, however, need to be considered when making design decisions.

Operating System

The second layer of the .NET Compact Framework architecture (cf. Fig. 3) 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. These differences affect elementary features such 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 (Sect. 5 presents a more detailed description of these issues):

- 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 [SNS03,MG00]: (1) class libraries, (2) execution engine, and (3) platform adaptation layer (cf. Fig. 3).

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 executing 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 port is based on a JIT compiler, not on an interpreter.

Because the design of the .NET Compact Framework anticipated operating system portability, access to core operating services occurs through a platform adaptation layer (PAL). The main responsibility of the 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.

5 PORTING THE .NET COMPACT FRAMEWORK

In the following, we describe our port of selected components of the .NET Compact Framework to the Symbian platform and discuss the major design decisions that affected our work. In order to do that, this section analyzes the characteristics of the Symbian operating system that caused most of the problems in our project, and proposes solutions for dealing with these issues.

We would like to point out that the result of our work is a research prototype, not a complete and thoroughly tested product. The main focus of our project was on evaluating whether it is feasible to port the .NET Compact Framework to Symbian phones by means of an exploratory approach. Having said this, the port runs stable and we were able to develop a range of neat applications based on our work.

C++ Dialect

The flavor of C++ that is 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 primarily introduced because of the limited device capabilities of Symbian smartphones. On the other hand, it appears that some of the restrictions and models of Symbian C++ still exist merely because of historical reasons and because of compatibility issues [EB04]. Important differences between Symbian C++ and standard C++ include: (1) different standard data types, (2) missing standard libraries such as a `libc`, (3) a special exception handling mechanism, and (4) a different memory management model.

First, the usage of simple data types such as `int` or `unsigned long` are not recommended by the Symbian Software Development Kit (Symbian SDK); so types such as `TInt` and `TUInt32` had to be used instead. Although these naming differences appear to be purely syntactic at first sight, they can result in hard to find errors when porting complex software such as the .NET Compact Framework CLR.

Second, as the full `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`). The STL (Standard Template Library) is also not supported in Symbian due to size limitations.

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 Symbian architects 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.

Finally, as mobile phones are switched on for long periods of time, the ability to reclaim all unused heap cells was crucial in the design of the Symbian operating system. 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 designed with memory-constrained 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 when executed. 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. In order to do this, 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 complete port of the .NET Compact Framework.

Executing .NET Portable Executables

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 to launch .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 of such servers are the file server, the socket server, and the window server. Servers are usually located in a different process or at least a different thread than the clients that want to access their services. The problem with

Symbian's client/server model 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 important 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_Write()` that sends a request to the Symbian file server to write data to a file. However, because only the client thread that created the connection to the file server can actually write to a file, the two .NET threads – which are both mapped to Symbian threads in our implementation – are not allowed to access the file. This problem persists independently of whether both threads want to write data to the same file or not; what matters here is only the fact that they want to access the file system. To solve this problem, we introduced a mediator thread that handles all communication with the file server. Symbian OS threads that represent application threads in .NET interact with this additional thread in order to access files. For the PAL implementation, this means that `PALFile_Write()` does not interact with the file server directly, but instead issues a request to the intermediary thread that communicates with the file server. A similar mechanism is used to handle networking and console access.

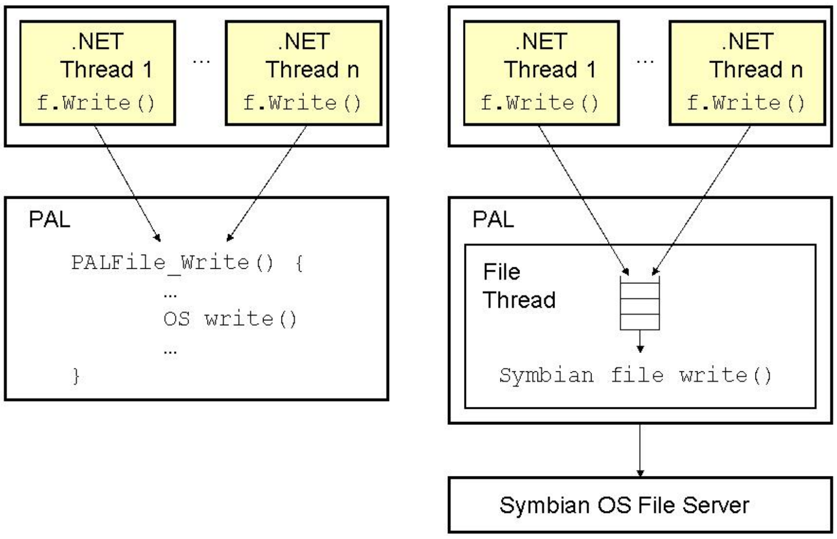

Figure 4: Implementing PAL functions on standard operating systems (left) and on Symbian (right).

Fig. 4 shows the effect of Symbian's client/server model and its security restrictions on the implementation of PAL functions. The left side of Fig. 4 shows how PAL functions on standard operating systems such as Windows or Unix simply map calls from the execution engine to the corresponding functions of the underlying operating system. In this model, the main task of PAL functions is data marshalling so that the parameters coming from the .NET framework are appropriately transformed into parameters for the operating system. In contrast, efficient access to Symbian OS functions (cf. right-hand side of Fig. 4) in the PAL requires an additional thread that sequentializes requests from different application threads. Besides the added complexity, another problem in this context is that many of the PAL functions must be carried out in an asynchronous way in order to deal with potential deadlock risks introduced by the sequentializing thread. Imagine, for example, two .NET threads that are used to write data to and retrieve data from a Socket; one .NET thread shall be responsible for sending and the other for receiving data. If now the receive function of the second thread is passed to the mediating PAL thread before the send is actually performed by the first .NET thread, and the receive call is implemented as a synchronous call (blocking), this will cause a deadlock since the send will never be performed. Hence, PAL functions in Symbian are not allowed to block, which again increases the complexity of the PAL implementation on this particular operating system.

Dealing with Symbian's Focus on Cooperative Multitasking

In the desktop domain, preemptive multitasking replaced cooperative multitasking years ago when resources became cheaper and PC-like systems much more computationally powerful. Furthermore, using preemptive 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 guarantee responsiveness [BRH90,JSM91]. However, although the Symbian operating system supports preemptive multitasking, switching between different preemptive threads is considered very expensive and programmers are strongly encouraged to use cooperative multitasking instead [EB04,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 [EB04,Har03] strongly suggest 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. Unfortunately, this 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 preemptively 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 sequentializing 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 preemptively scheduled threads on the Symbian operating system layer.

GUI

Symbian allows access to the GUI at 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 graphics libraries, the most common being the AVKON library built for Series 60 phones.

Four distinct approaches were identified that can be followed when implementing the GUI:

- *Using a portable graphics library to create a System.Windows.Forms-compliant GUI for Symbian smartphones.* The main advantage of this approach is that by using a portable graphics library the .NET GUI interface does not rely on the specific user interface platform of a particular smartphone model. Consequently, the GUI can be easily deployed on different Symbian phone models such as Series 60 devices and Ericsson phones. On the other hand, the look-and-feel is different from native Symbian applications, which might reduce usability.
- *Mapping .NET user interface calls to AVKON or alternative UI libraries.* 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 from accessing the same control or controls from having a parent-child window relationship.
- *Providing access for device-specific GUI libraries such as AVKON.* 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. A major problem of this approach is also that a GUI application cannot be deployed on other devices supporting the .NET Compact Framework, such as Windows-based smartphones.
- *XML-based GUI abstractions.* It is possible to create an abstraction for interface designers at a higher level [APP99]. In the Views GUI engine [BH05,Mil05], for example, an XML-based notation simplifies the implementation of user interfaces in Rotor across different operating systems.

Considering the above alternatives, the major design decision when implementing a .NET GUI for Symbian smartphones is whether to base the user interface upon the controls provided by the underlying operating system or not. As already mentioned in Sect. 3, we have decided to remain in full control of the drawing process by using custom controls that are similar to the ones on Windows-based smartphones. The main advantage of this approach is that we increase the portability of GUI applications across different .NET platforms because we can expose the standard APIs of System.Windows.Forms to application programmers. An important implication of this is that all the standard programming tools that are available for .NET application development – as for example the Form Designer that is integrated into Visual Studio –

can be used for application development on Symbian as well. I.e., an application developer can graphically arrange the GUI controls she needs in her program as well as implement the standard delegates that are assigned to GUI events using the tools integrated into Visual Studio. Afterwards she can then deploy the resulting application without any modifications on a Windows- as well as on a Symbian-based smartphone.

As mentioned above, a potential problem with our approach is reduced usability because of performance issues and because the controls of .NET applications differ from the ones provided by native smartphone applications. With respect to performance, Sect. 6 shows that our implementation provides very good responsiveness. In fact, in our demo applications there is no apparent difference in performance to native GUI applications. A problem that remains is that our GUI controls differ from those of Series 60 and UIQ. On the other hand, Windows-like controls expose a very good usability, and many phone owners know how to handle Windows-like controls from their PCs at home or at work. We therefore think that GUI applications implemented using System.Windows.Forms provide an easy to handle user interface for owners of Symbian smartphones.

6 EVALUATION

The purpose of this section is to provide a detailed performance evaluation of our implementation. The evaluation consists of four main parts: (1) an overview of the memory requirements of our port, (2) a performance analysis of basic services such as method calls in comparison to a Java Virtual Machine, (3) an analysis of the performance penalties caused by Symbian's programming model, and (4) a performance analysis of our GUI implementation in comparison to native drawing primitives.

We have to emphasize that the results shown here assess only the performance and memory requirements of our port. This port was done as a means to evaluate the feasibility of porting the .NET CF to Symbian devices. Hence, the following figures should not be used as a general comparison between Java and .NET (for example, [Sin03] provides a technical comparison of the different virtual execution systems).

Memory Requirements

Sect. 3 reviewed the hardware characteristics of today's Symbian smartphones and identified the low amount of available memory as one of the main constraints of this device platform. While certain methods can be used to reduce the amount of RAM required by a virtual execution system, the amount of non-volatile memory necessary to host our current implementation is as follows:

First, the size of the non-managed part of the virtual execution system is 1.6 MB. This includes the JIT compiler, the garbage collector, and the entire PAL implementation. The 1.6 MB also includes most of the GUI code. (Note that our figures are only valid for our port; they do not apply for the commercially available .NET Compact Framework.) If no GUI is required, the corresponding DLL is of course not needed on the actual device. It should also be kept in mind that the performance of the GUI on resource-constrained device platforms is of central importance, so that most of the GUI functionality has to be implemented in native code. Consequently, the System.Windows.Forms managed code library primarily contains mappings to the portable windowing toolkit and the underlying graphics library. It is also noteworthy that because of Symbian's DLL model (no writable static variables) native code can be designed in such a way that only a small amount of the 1.6 MB has to be loaded into RAM. In our current prototype, however, all of the 1.6 MB is loaded into RAM when a .NET application is executed on a Symbian smartphone.

Second, the size for managed code libraries is the same as on the standard .NET Compact Framework for Windows, except for the GUI libraries System.Drawing and System.Windows.Forms. The reason for this is that – except for the GUI – we could leave all managed code untouched. In fact, this was one of our main design goals because it ensures that we expose exactly the same programming model on Symbian devices as on Windows smartphones. The amount of memory for managed code libraries depends on the desired functionality. To get a text-based application up and running together with file and networking functionality requires 882 KB of managed libraries. These libraries are loaded into RAM in our implementation. The managed GUI libraries take another 90 KB at the moment (remember that most of the GUI functionality is in native code). The biggest chunk of memory is required by the managed code XML library, which is 1.16 MB in size. This library is necessary for Web Services support.

In summary, our current implementation requires 3.7 MB of flash memory for all the functionality we have discussed in this paper. Regarding the size of flash memory on newer smartphone models, this should not be a problem. Newer phone models also have ever more sophisticated cameras integrated, and therefore an increasing demand for flash memory to store pictures and applications. In this context, 3.7 MB of flash for a virtual execution system including managed libraries seems reasonable. In the current version of our framework, all necessary libraries are loaded into RAM, which is a waste of resources. Consequently, although 3.7 MB is a very small size for an execution engine it still leaves much room for improvements. This is a work item that we want to address in the future.

Performance Analysis of Basic Services

In the following, we compare the time necessary to execute .NET code on our platform with the time needed to execute corresponding Java code on Symbian smartphones. As it would be 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.

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 [jMocha]. 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 so 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).

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. 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.

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

Table 2: Time for running benchmarks (in ms).

Tab. 2 shows the results of running the tests. The first micro-benchmark in our evaluation measures memory read latency by accessing the elements of arrays. The test varies the size of the arrays as well as the patterns in which elements are read [jMocha]. 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. To some degree, this test also illustrates the performance of the memory subsystem. 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.

The first column of Tab. 2 shows the name of the micro-benchmark. The second lists the parameters used to run the micro-benchmark. 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. 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 two different phones (Phone A and Phone B) and a standard PDA in our experiments. Phone A runs Symbian OS version 6.1, has 3 MB available memory, and a 104 MHz processor, while Phone B runs Symbian OS version 8.0a, and has 10 MB of available memory and a 220 MHz processor. The PDA is a T-Mobile MDA II running PocketPC 2003. 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 port of the .NET Compact Framework is comparable to 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 cases correspond to tests calling synchronized 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 methods is 157 times slower than an unsynchronized method call on Phone A. We expected calls to a synchronized method to be only 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 is responsible for the poor performance. 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. Spawning a thread is also considerably slower in our Symbian .NET

Compact Framework implementation than in the Java implementation. These problems have been solved in newer releases of the .NET Compact Framework.

Performance Impact of Symbian Constraints

In Sect. 5 we discussed the implications of Symbian's programming model on the implementation of CLI-compliant virtual execution systems. One of the core problems in this context was that additional threads had to be introduced in the PAL in order to sequentialize requests from different .NET application threads. As the usage of preemptively scheduled threads in Symbian is strongly discouraged, we evaluated the impact of additional threads in the PAL (cf. Fig. 5 and Fig. 6). As a result it can be said that the performance penalty caused by the sequentializing PAL thread depends on the complexity of the .NET function that has to be mapped to the underlying operating system. If its complexity is high, the performance penalty is acceptable because the overhead caused by additional thread switches is small compared to the time needed to carry out the actual function. For average operations, however, the performance penalty can be significant. Fig. 5 shows this for a function that puts the cursor to a specified location on the console window. If the requests from the execution engine are passed through the additional thread (upper curve in Fig. 5) this causes a performance penalty of around 20 %.

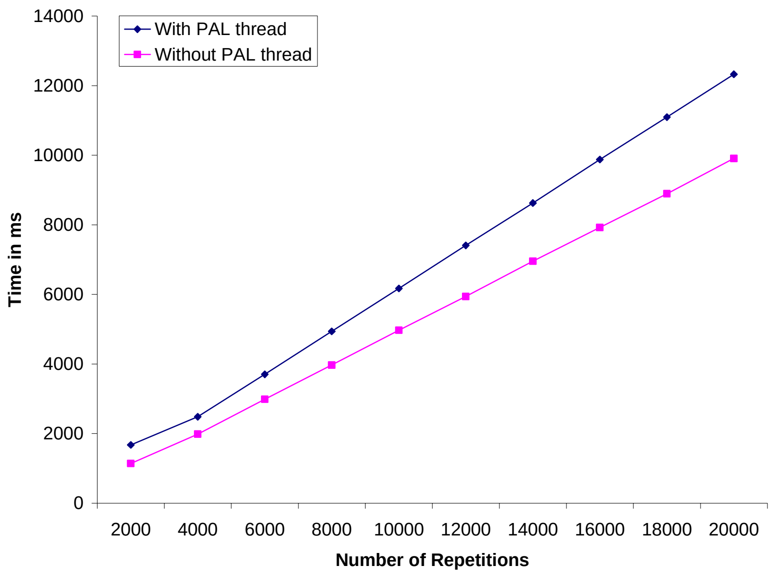

Figure 5: Performance penalty of sequentializing PAL threads.

As the usage of preemptively scheduled threads seems to be problematic in Symbian, another experiment was carried out in which we tried to measure the impact of preemptively scheduled threads. In the experiment we created up to 31 threads that repeatedly called a single function, and measured the performance of carrying out 100'000 function calls. The operating system distributed these calls equally so that each thread issued approximately the same amount of calls. In the case of 10 threads, each thread would invoke the function around 10'000 times. As can be seen in Fig. 6, the amount of threads does indeed have a significant impact, which is different from a

Windows environment. On the emulator, for example, which runs in a Windows environment, the curve is flat. The impact of threading on system performance can be quite different between operating systems [ZY98]. Fig. 6 does also compare the performance difference between native and managed threads. As expected, managed threads (see upper curve in Fig. 6) cause a nearly constant performance overhead.

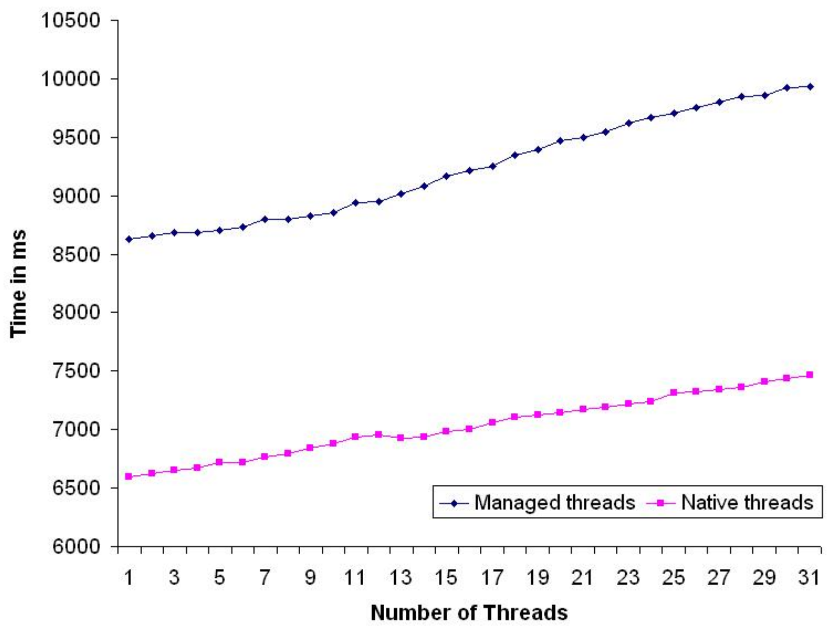

Figure 6: Performance difference between managed and native threads.

GUI Performance

A main problem of many GUI implementations that do not rely on the GUI controls provided by an underlying operating system is bad performance and insufficient responsiveness. As our GUI model is based on a portable graphics library and windowing toolkit (and does not use Symbian controls), this subsection is devoted to analyzing the performance of our portable .NET GUI for Symbian smartphones.

Generating a portable UI on a smartphone is a two-step process. First, the controls and other UI elements are drawn to an off-screen bitmap. In a second step, the off-screen bitmap must be copied to the smartphone's screen. Both of these processes take place in unmanaged code. In order to evaluate the performance of the drawing process itself, Fig. 7 compares Symbian's `DrawLine` function with the implementation in our portable graphics library. As can be seen, both implementations are comparable. The performance penalty of the Symbian `DrawLine` function is probably a result of Symbian's client/server model. The result of this is that drawing a line would internally require interaction with a Symbian server that is responsible for drawing. This assumption is supported by the fact that the performance of more complicated Symbian drawing functions (rectangles, text, etc.) is usually better than our implementation. Nevertheless, the .NET GUI implementation for Symbian smartphones has a very good performance.

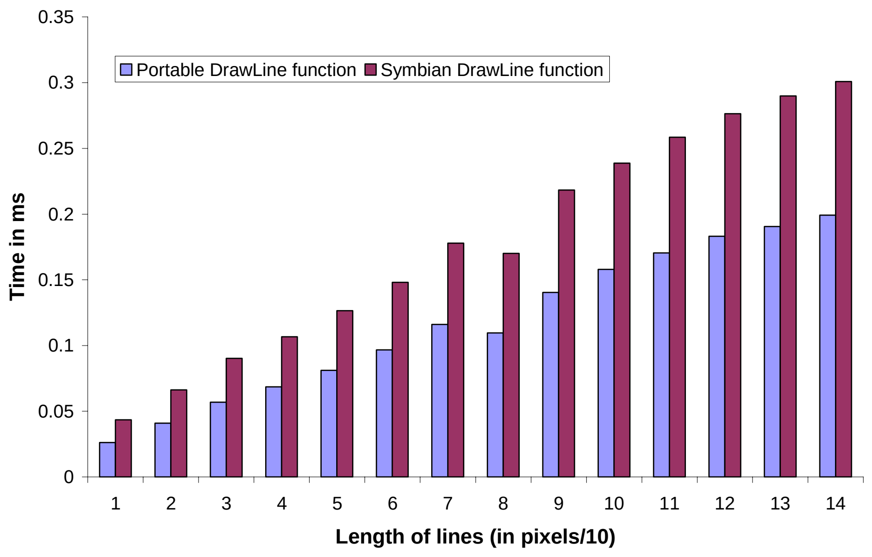

Figure 7: Performance of the DrawLine function in Symbian and in our portable graphics library.

Fig. 8 compares different methods for copying the generated off-screen bitmap to the phone's display. Not surprisingly, this process is very time consuming if the off-screen bitmap is copied bit by bit to the display using Symbian's `plot` function (see left bar in Fig. 8). When considering the internal format in which Symbian is storing bitmap data – no conversion between different data formats is then necessary – the transfer to the phone's display can be very fast. According to our measurements this takes about 5 ms. We were also interested in the difference between direct screen access and drawing to a Symbian window. As can be seen in Fig. 8, the difference between both approaches is negligible so that we draw to a Symbian window in our implementation instead of using direct screen access. This has certain advantages with respect to event handling.

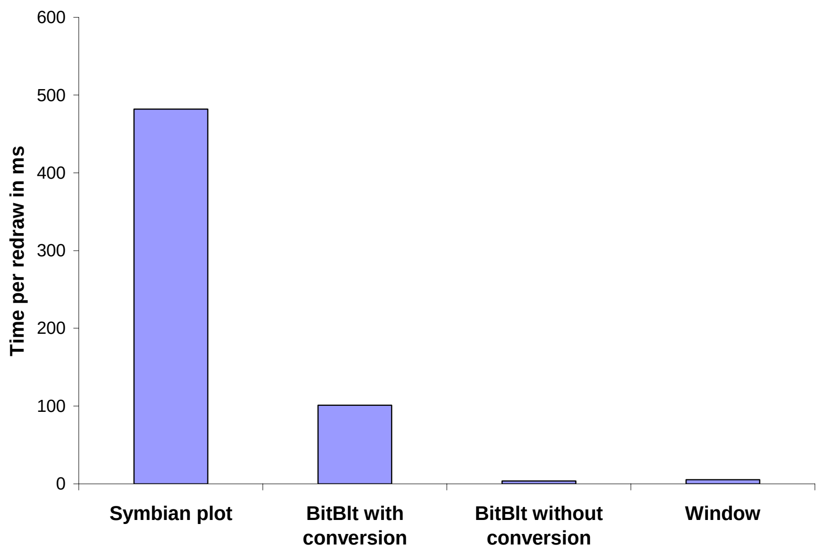

Figure 8: Copying the off-screen bitmap to the smartphone display.

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. The gap between the programming model of .NET and that of Symbian – e.g. its focus on cooperative multitasking and its local client/server model – were the main obstacles in our project. Our work also proves it possible to realize a portable .NET GUI on Symbian smartphones that conforms to System.Windows.Forms and offers good responsiveness. Last but not least, we verified that the performance of carrying out .NET applications on Symbian is comparable to that of Java applications.

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 their helpful comments and their continuous support throughout this project. We also would like to thank the Microsoft .NET Compact Framework team and the anonymous JOT reviewers.

REFERENCES

[AP99] Abrams, M.; Phanouriou, C.; Batongbacal, L.; Williams, S.; Shuster, J.: "UIML: An Appliance-Independent XML User Interface Language", 8th International World Wide Web Conference, Toronto, Canada, May 1999.

[BH04] Bishop, J.; Horspool, N.: "Developing Principles of GUI Programming using Views", 35th SIGCSE Technical Symposium on Computer Science Education, pp. 373-377, Norfolk, USA, March 2004.

[BKR04] Benton, N.; Kennedy, A.; Russo, C.: "Adventures in Interoperability: The SML.NET Experience", 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, pp. 215-226, August 2004.

[BLWG99] Bisbal, J.; Lawless, D.; Wu, B.; Grimson, J.: "Legacy Information Systems: Issues and Directions", IEEE Software, vol. 16, no. 5, pp. 103-111, 1999.

[BRH90] Baruah, S.; Rosier, E.; Howell, R.: "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor", Journal on Real-Time Systems, vol. 2, no. 4, pp. 301-324, November 1990.

[BW05] Bishop, J.; Worrall, B.: "Towards platform interoperability: retargeting a GUI library on .NET", 3rd International Conference on .NET Technologies, Plzen, Czech Republic, May 2005.

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

[Canalys] Analysis for the high-tech industry, http://www.canalys.com.

[CLI] ECMA and ISO/IEC C# and Common Language Infrastructure Standards, http://msdn.microsoft.com/netframework/ecma/.

[EB04] Edwards, L.; Barker, R.: *Developing Series 60 Applications, A Guide for Symbian OS C++ Developers*, Addison-Wesley, 2004.

[Forge] AppForge, http://www.appforge.com.

[Gut01] Gutknecht, J.: "Active Oberon for .NET: An Exercise in Object Model Mapping", First Workshop on Multi-Language Infrastructure and Interoperability, September 2001.

[GW04] George, M.; Wong, W.: "Windows CE for a Reconfigurable System-on-a-Chip Processor", International Conference on Field-Programmable Technology 2004, pp. 201-208, December 2004.

[Ham03] Hamilton, J.: "Language Integration in the Common Language Runtime", ACM SIGPLAN Notices, vol. 38, no. 2, pp. 19-28, February 2003.

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

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

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

[JSM91] Jeffay, K.; Stanat, D.; Martel, C.: "On non-preemptive scheduling of periodic and sporadic tasks", 12th Real-Time Systems Symposium, pp. 129-139, December 1991.

[MG00] Meijer, E.; Gough, J.: "Technical Overview of the Common Language Runtime", Microsoft white paper, 2000.

[Mil05] Miller, D.: "Mirrors: a system for GUIs using reflection in .NET 2.0", Technical Report, University of Pretoria, South Africa, September 2005.

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

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

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

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

[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.

[RV01] Riggs, R.; Vandenbrink, M.: *Programming for Wireless Devices with the Java 2 Platform Micro Edition*, Addison-Wesley, 2001.

[Sin03] Singer, J.: "JVM versus CLR: a comparative study", 2nd International Conference on Principles and Practice of Programming in Java, pp. 167-169, Kilkenny City, Ireland, June 2003.

[SNS03] Stutz, D.; Neward, T.; Shilling, G.: *Shared Source CLI Essentials*, O'Reilly, 2003.

[SSM00] Simon, R.; Stapf, E.; Meyer, B.: "Full Eiffel on the .NET Framework", Microsoft Developer Network, July 2002.

[ZY98] Zabatta, F.; Ying, K.: "A Thread Performance Comparison: Windows NT and Solaris on a Symmetric Multiprocessor", 2nd USENIX Windows NT Symposium, Seattle, WA, 1998.

About the authors

Frank Siegemund received a PhD degree in Computer Science from ETH Zurich and is currently a software design engineer at the European Microsoft Innovation Center. His research interests lie in the areas of virtual execution systems for mobile and embedded devices, wireless communication protocols, and handheld computing. He can be reached at franksie@microsoft.com.

Robert Sugar holds a Master's Degree in Computer Science from the Budapest University of Technology and Economics. Currently he is a PhD candidate in Computer Science and works as a software design engineer in the European Microsoft Innovation Center. He can be reached at rsugar@microsoft.com.

Alain Gefflaut is a software design engineer at the European Microsoft Innovation Center. His research interests are in the areas of networking protocols, mobile devices and operating systems. He holds a PhD and a Master's degree, both in Computer Science, from the University of Rennes in France. He can be reached at alaingef@microsoft.com.

Friedrich van Megen graduated as a Diplom-Informatiker from the University of Kaiserslautern. He joined the European Microsoft Innovation Center as a software design engineer. His research interests include mobile devices and context-aware computing. He can be reached at fmegen@microsoft.com.