.NET Runtime and Tools for Tizen Operating System

Alexander Soldatov(✉), Gleb Balykov(✉), Anton Zhukov(✉), Elena Shapovalova(✉), and Evgeny Pavlov(✉)

Samsung Research Russia, Moscow, Russia
{soldatov.a,g.balykov,a.zhukov,elena.sh,e.pavlov}@samsung.com
https://research.samsung.com/srr

Abstract. Samsung Electronics and Microsoft Corporation have been developing open source implementation of .NET platform called .NET Core since 2016. This platform is usually used for implementation of server-side and desktop applications, and Samsung has also adopted .NET Core virtual machine and libraries for Tizen OS. This solution was integrated into various ARM CPU based devices e.g. smart watches, TVs and other electronic devices. Tizen has always supported variety of languages and SDKs for developers. .NET has greatly expanded this variety by supporting new tools and new languages. This paper describes major challenges that we have encountered during integration of .NET to Tizen OS as well as optimizations, which were applied to .NET Core to make applications startup and memory consumption better on variety of devices.

Keywords: Managed programming language · Compiler · Language runtime · .NET

1 Introduction

Throughout the long history of Tizen [5], variety of different technologies and languages were available to application developers. Initially developers were provided with native SDK with C/C++ support. This solution has demonstrated good performance and memory consumption of applications, but currently C/C++ languages are not very popular anymore because of difficulty of development. They require long development cycle and provide poor memory safety.

Web applications propose another approach to applications development on Tizen. While JavaScript and HTML5 gained popularity, this approach became very attractive. It allows to create applications very easily and fast. However, this approach is not effective in terms of hardware resources usage, specifically, startup time and memory consumption. Native applications are much better optimized and customized in comparison with web-applications. These metrics

Published by Springer Nature Switzerland AG 2020
V. Ivanov et al. (Eds.): OSS 2020, IFIP AICT 582, pp. 190–195, 2020.
https://doi.org/10.1007/978-3-030-47240-5_19

are essential for consumer electronics, thus web-applications can not be chosen as the most satisfying development solution.

.NET Core [6] and C# language mitigate problems of the two approaches described above. They provide reach set of features and frameworks for application developers and demonstrate good performance and memory consumption. Table 1 shows comparison for different languages and technologies on the Samsung Galaxy Gear Watch 2 with Tizen 5.0 and Stopwatch application.

Table 1. Comparison of Tizen developer technologies

Technology/Function	Native	Web applications	.NET Core
Memory usage	8.6 MB	65.4 MB	27.3 MB
Application startup time	0.4 s	1.2 s	1.0 s
Memory safety	No	Yes	Yes

Moving an ecosystem to a new operating system and CPU requires a large amount of work and optimization of existing code base. For example, the following parts of virtual machine should be updated or created from scratch: JIT compiler, low level code of virtual machine, which deals with operating system, parts of frameworks interacting with hardware. Furthermore, development tools like debuggers and profilers should also be created for non-supported operating system. All this work has been done for Tizen and ARM CPU, as well as Tizen specific performance optimizations, which allowed to deploy .NET Core on devices with small amount of memory and low- and mid-level CPUs. The details of those optimizations will be described below.

The goal of this paper is to describe Tizen .NET implementation specifics, both of virtual machine and developer tools. Section 2 describes high-level Tizen .NET architecture, with details of .NET Core virtual machine architecture. Section 2.1 includes high-level overview of .NET Core and Tizen optimization details. Section 2.2 describes Tizen-specific development tools.

2 Architecture

Figure 1 shows high level architecture of Tizen .NET platform. All three types of application development technologies used with Tizen rely on native subsystems, giving applications access to different parts of the device. `CoreCLR` virtual machine [1] and `CoreFX` basic libraries [2] are the foundation of the .NET ecosystem in Tizen. They interact with operating system directly and allow to execute user's code. `TizenFX` and `Xamarin.Forms` are situated on the next level. These are the frameworks which provide additional functionality for developers and allow them to create user interfaces and utilize platform specific functionality from C# code.

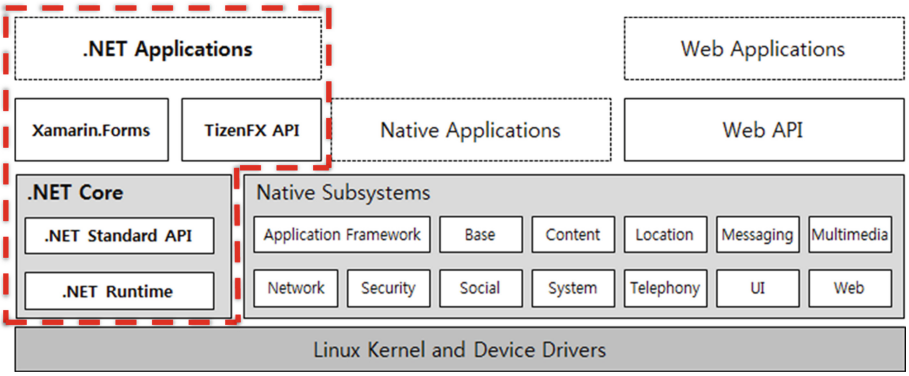

Fig. 1. Tizen .NET architecture

Tizen uses special launchers for different types of applications. .NET specific launcher is called `dotnet-launcher`. It performs initialization of CoreCLR and some additional activities (that will be described below) inside a `candidate process` before the application to be launched is determined. When request to launch application comes, the `candidate process` with prepared runtime already exists.

2.1 .NET Runtime

Main part of work related to enabling of .NET Core on Tizen was performed in runtime, specifically, in JIT compiler. JIT translates stack-based Microsoft Intermediate Language (MSIL) bytecode to the target CPU machine code and applies various optimizations. RyuJIT was chosen as a compiler, but it didn't support ARM. As this support is required for Tizen, it had to be implemented from scratch.

During this work all of the stages of compilation, that precede Code Generation, had to be updated to support ARM. Then some performance and memory consumption optimizations were applied. .NET Core versions differ in performance and memory consumption, thus optimizations applied were not universal.

Removal of Relocations. This optimization reduces memory consumption. It turns out that some applications use the same libraries while being launched. So CoreCLR loads system libraries (DLLs) using `mmap` system call. Then those libraries may be shared between the applications. These DLLs are read-only, but they can be compiled to native images. Native images contain relocations, i.e. data or code, which rely on the actual starting memory address of mmapped file. This data or code should be updated at DLL load time, creating process specific copies of memory pages. The idea of optimization is to reduce number of such relocations making memory pages position independent and read only and, thus, shared between processes. This has been performed for various parts

of native images and reduced CoreCLR Private memory consumption of each process by 45% (3 Mb) in average.

Direct Mapping of Assemblies. Previously, plain DLLs with section alignment smaller than a size of a virtual page had to be copied to anonymous memory. This means that each process will contain its own copy of DLL in physical memory. After optimization DLLs without writable sections (this is the case on Tizen) are directly mmaped to the memory. This is the optimization, which made managed DLLs shareable between processes in the first place. After this optimization CoreCLR Private memory consumption of each process was reduced by 9% (645 Kb) in average.

Simplification of Allocators. CoreCLR has many different allocators in different parts, and each allocator serves its purpose. One of the optimized allocators has been simplified to plain `malloc/free` allocator instead of the allocator with default prealloced memory, and this had no effect on performance. The other one (arena allocator) is used by JIT during compilation and never frees its memory by default. This behavior was changed to basically the same `malloc/free` and also had no negative effect on performance. Optimization of the former reduced CoreCLR Private memory of the application process by 11.5% (776 Kb) in average, optimization of the latter reduced Private memory of the whole application process by 8% (1.2 Mb) in average.

Compact Entry Points. CoreCLR uses small thunks of code to perform call indirection, for example, to trigger compilation of method when it is called for the first time. This way a unique entry point is assigned to each method. There are different types of these thunks, e.g. `FixupPrecode`, `StubPrecode` or `Compact` entry points. Compact entry points are designed to occupy as little memory as possible by combining some of the auxiliary data of few methods (see Listing 1). Each compact entry point on ARM occupies 4 bytes and `CentralJump` is 14 bytes. In comparison, each `FixupPrecode` occupies 12 bytes and its equivalent to `CentraJump` is 4 bytes.

Listing 1. Compact entry points

```
entry1:
  mov r12, pc
  b CentralJump
...
entryN:
  mov r12, pc
  b CentralJump
CentraJump:
  ldr pc, pThePreStubCompactARM
  nop
  dw pChunk
  dw pThePreStubCompactARM
```

Previously, `Compact` entry points were not implemented for ARM, and their implementation reduced CoreCLR Private memory of the process by 2.2% (80 kb) in average. Besides, some micro benchmarks showed 15% improvement of performance.

CoreCLR Initialization in Candidate Process. Previously CoreCLR was initialized after the application launch request is sent to `dotnet-launcher` (launcher of C# applications). It consumed startup time observed by users. To improve this point, CoreCLR initialization is now performed before application launch request is received (see Fig. 2). Runtime becomes prepared to launch actual application, mitigating runtime initialization time.

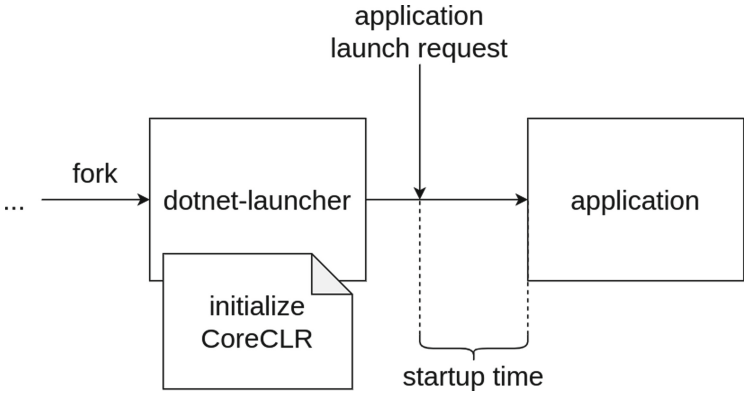

Fig. 2. CoreCLR initialization in candidate process

Class Preloading. Since runtime is now initialized prior to the application launch request and there is some time before this request will actually come in, we can do some additional initialization. The idea of this optimization was to load some common DLLs and C# classes from libraries to memory even before the application launch request. When the class loading for application is performed, some of the classes will already be loaded to memory, thus startup time will be smaller. This optimization reduced startup time of applications by 3.7% in average.

Hydra Launcher. We can go even further with the work we do prior to application launch request. In case the Linux process mmaps native images to memory, then applies relocations and then performs fork, all these dirty memory pages move to the Shared part of memory of all child processes, i.e. only one copy of these memory pages will exist in physical memory, thus reducing memory consumption. By applying the same technique and creating additional parent process called Hydra, which mmaps native images of system DLLs to memory and applies relocations to them, we were able to reduce memory consumption of the whole application process by 14% in average.

2.2 .NET Tools

This section provides a brief review on .NET tools provided by Samsung. Visual Studio Plugin for Tizen [3] is one of major tools for Tizen developers. This plugin includes two components: NetcoreDBG [4] (debugger) and Performance profiler [7]. Both these components were developed from scratch as open source projects.

Samsung NetcoreDBG Debugger. This debugger was developed as an alternative managed debugger for .NET applications. Microsoft provides *proprietary debugger* (VSDBG) for this purpose, but it sets several limitations: it is compatible with VSCode IDE or Microsoft Visual Studio only; it does not support ARM architecture. NetcoreDBG overcomes these limitations: it may be used in several OS-es (MS Windows, Linux, Tizen, Mac OS) and it supports various architectures (x86, ARM, x64).

Samsung C# Profilers. The two major profiling opportunities are performance and memory profiling. The data gathered by profilers is processed and presented in a graphical interface of VS plugin.

The performance profiling mode shows real-time memory usage as well as CPU load. Also it allows to track JIT and GC events depending on the profiling options.

Memory profiler shows amount of dynamically allocated memory by user code and by runtime itself. The allocated memory is classified as Heap Allocated, Heap reserved, unmanaged memory and Span (memory allocated by data type during application lifespan).

3 Conclusion

This paper describes major challenges that we have encountered during integration of .NET on Tizen operating system as well as optimizations, applied to .NET Core to make applications startup and memory consumption better. The next plans are related to the support of new versions of .NET Core and implementation of new ARM-specific optimizations for the newest Tizen releases.

References

1. CoreCLR github. https://github.com/dotnet/coreclr
2. CoreFX github. https://github.com/dotnet/corefx
3. Samsung Visual Studio Tools for Tizen. https://github.com/Samsung/vs-tools-cps
4. Samsung NetcoreDBG last releases. https://github.com/Samsung/netcoredbg/releases
5. Samsung Tizen official. https://www.tizen.org/
6. Microsoft .NET official. https://dotnet.microsoft.com/
7. Samsung .NET Performance Profiler manual. https://github.com/Samsung/vs-tools-cps/blob/master/docs/tools/profiler-user-manual.md