MonoRT and Benchmark Manual

Martin Däumler

1 Introduction

This document describes the build instructions and the usage of the modified Mono VM “MonoRT” on a Linux system. It also explains the MonoRT-internal experiments and the micro benchmarks. The source code of MonoRT and of the micro benchmarks are part of the archive, which contains this manual. MonoRT’s source code is in directory “mono-rt”, the micro benchmark source code and results are in directory “comparative_experiments” and the internal experiments’ files are in directory “internal_experiments”.

2 MonoRT

2.1 Build MonoRT

General instructions how to build Mono and MonoRT, respectively, from source code can be obtained from [1]. On ARM systems, it has to be ensured that line 2556 of file “configure.in” is set to `mono_debugger_supported=no` and that `autoconf` and `automake` are run before starting the configuration. The version of Mono, which is used for MonoRT, has no prober support for the debugger on ARM, so that these steps ensure that the debugger support is not built.

It is recommended to configure MonoRT by issuing the command:

```
$ autoconf
$ automake
$ ./configure --prefix=/usr/local
```

Missing packages have to be installed on the system. MonoRT is built and installed by issuing the commands

```
$ make
$ make install
```

The latter might require super user rights. Check if the installation of MonoRT succeeded by issuing the following command and compare the output to that:

```
$ mono --version
Mono Real-Time based on Mono JIT compiler version 2.6.1
Copyright (C) 2002-2008 Novell, Inc and Contributors.
www.mono-project.com
```

2.2 MonoRT usage

Mono's and MonoRT's, respectively, command line parameters can be obtained from Mono's man page. Additional command line parameters, which are associated to real-time code generation, are listed by Table 1.

parameter	**description**	**support on IA32**	**support on ARM**
--prejit	enable pre-compilation only	X	X
--prepatch	enable Pre-Patch and pre-compilation	X	
--jtrace	enable tracing of JIT compiler activity at runtime (not recommended in JIT mode)	X	X
--ptrace	enable tracing of reference resolution at runtime (not recommended in JIT mode)	X	X
--wtrace	enable tracing of helper function generation and class loading at runtime (not recommended in JIT mode)	X	X
--wait	enable 30 sec. countdown after pre-compilation and Pre-Patch for use of Checkpoint/Restore tool	X	X

Table 1: Additional command line parameters of MonoRT

The tracing functionality is useful for verification of the real-time code generation. These parameters are demonstrated on a simple Hello-World application. The application is compiled with Mono's C# compiler `mcs`. It is recommended to include debug symbols in order to get meaningful back traces. Section 4.3.2 of the thesis describes that the pre-compilation and Pre-Patch are triggered via managed code, which is realized as library, in order to get a proper exception handling. The source code of this library is located in subdirectory "rttools" of MonoRT's source directory. The library `MonoRT.dll` is build by the following command:

```
$ mcs -debug+ MonoRT.cs -t:library
```

This command builds the library and its debug symbols (`MonoRT.dll.mdb`), which are necessary for meaningful back traces. The library `MonoRT.dll` has to be located in the working directory of the application that uses real-time code generation functionality. If the library is not available, the real-time code generation does not work. Building and running the application in JIT-based pre-compilation mode generates the following output:

```
$ mcs -debug+ hello.cs
$ mono --prejit --debug -O=-aot hello.exe
Pre-compilation starts
  Pre-compile    /home/dev/hello.exe
  Pre-compile    /usr/local/lib/mono/1.0/mscorlib.dll
Pre-compilation finished
```

```
Hello World!
```

Running the application in JIT-based Pre-Patch mode generates the following output:

```
$ mono --prepatch --debug -O=-aot hello.exe
Pre-compilation starts
  Pre-compile    /home/dev/hello.exe
  Pre-compile    /usr/local/lib/mono/1.0/mscorlib.dll
Pre-compilation finished

Pre-Patch starts
Pre-Patch finished

Hello World!
```

Running the application in AOT-based Pre-Patch mode with tracing enabled generates the following output:

```
$ mono --aot /home/dev/hello.exe
$ sudo mono --aot /usr/local/lib/mono/1.0/mscorlib.dll
$ mono --prepatch --debug --jtrace --ptrace --wtrace
-O=aot hello.exe
Pre-compilation starts
  Pre-compile    /home/dev/hello.exe
  Pre-compile    /usr/local/lib/mono/1.0/mscorlib.dll
Pre-compilation finished

Pre-Patch of PLT starts
  /usr/local/lib/mono/1.0/mscorlib.dll.so check sum: 129
  /home/dev/hello.exe.so check sum: 240
Pre-Patch of PLT finished
Pre-Patch starts
Pre-Patch finished

Hello World!
```

In case of AOT-based Pre-Patch, the AOT compiled images are listed for supervisory purposes. There are no tracing outputs in the listing above. This indicates that there are no code generation activities at runtime.

Running the application with pre-compilation only and tracing of reference resolution enabled produces output that is similar to the following (output shortened):

```
$ mono --prejit --debug --ptrace -O=-aot hello.exe
Pre-compilation starts
  Pre-compile    /home/dev/hello.exe
  Pre-compile    /usr/local/lib/mono/1.0/mscorlib.dll
Pre-compilation finished
[...]
  Mono-RT - mono_magic_trampoline ()
        MONO_VTABLE_METHOD-Pfad
                Klasse: SynchronizedWriter
                VTable-Displacement: 14
        Methode: System.IO.SynchronizedWriter:WriteLine (string)
        Jump: nein
        Thread: 2631408
        AppDomain: 0
   at System.Console.WriteLine(System.String value)
     in /.../corlib/System/Console.cs:414
   at System.Console.WriteLine(System.String value)
     in /.../corlib/System/Console.cs:413
   at HelloWorld.Hello.Main() in /home/dev/hello.cs:8
[...]
```

That is, if a reference is resolved at run time, a stack trace is also given in order to find the cause. In the example, line 8 of of the source file leads to the resolution of a reference.

3 Internal Experiments

3.1 Preparation

The Mono-internal runtime test suite [2] is used to evaluate the functional correctness of MonoRT. Robert Andre provided a modified set of driver files of Mono's test suite, which allows running certain tests separately and with customized command line parameters. These files are located in the directory "internal_experiments". They have to be copied into MonoRT's source directory. Note, existing files are replaced, so a backup is recommended.

It is necessary to limit the test cases for the verification of the real-time code generation to test cases that can be compiled by Mono's C# compiler `mcs`, which supports the .NET-1.x-profile only. For that purpose, subdirectory "mini" of directory "internal_experiments" includes a customized version of the file `TetsDriver.cs`, which is named `TestDriver-1.0.cs`. It has to be re-named to `TestDriver.cs` and copied to subdirectory "mini" of MonoRT's source directory. An analogous procedure has to be applied to the file `Makefile-monort-10` in subdirectory "test" of directory "internal_experiments". There are also files for tests without real-time code generation that are not restricted to the `mcs` compiler. They are named `TestDriver-2.0.cs` and `Makefile-monort-20`, respectively.

3.2 Run Internal Experiments

Runtime tests, which use the custom driver files, are run in subdirectory “tests” by issuing the following command:

```
$ ./doTests -mkt runtest
[...]
380 test(s) passed. 0 test(s) did not pass.
```

In this case, the entire runtime test suite was run, which includes 380 test cases. In order to run the test cases, which are supported by the .NET-1.x-profile and the `mcs` C# compiler, respectively, the files `TestDriver-1.0.cs` and `Makefile-monort-10` have to be copied like described above. Runtime tests with pre-compilation enabled are run by issuing the following command:

```
$ ./doTests --prejit -mkt runtest
[...]
259 test(s) passed. 2 test(s) did not pass.
```

Section 7.1.3 of the thesis describes that the two test cases `classinit.exe` (number 157) and `bug-82022.exe` (number 250) fail due to side effects of type constructor and exception handling execution order, respectively. Both test cases remain in the test suite, because they address an issue, which might be handled in future versions of MonoRT.

Test cases can be executed individually by passing the addional parameter `-t` followed by the range of the test case number. For example, the following command runs the test case number 5 to 7 with Pre-Patch, JIT compiler tracing and reference resolution tracing enabled:

```
$ ./doTests --prepatch --jtrace --ptrace -mkt runtest
-t 5-7
[...]
1 test(s) passed. 0 test(s) did not pass.
```

Several test cases might produce tracing output, which indicates native code generation and modification at run time. Each of these cases is traceable to the limitiations, which are described in section 7.1.3 of the thesis. For example, test case number 4 (`assemblyresolve_event3.exe`) has the following tracing output (shortened):

```
$ ./doTests.sh --prepatch --jtrace --ptrace -O=-aot
-mkt runtest -t 4-4
test_to_run: 4
[4]:
Testing assemblyresolve_event3.exe...
[...]
Mono-RT - mono_delegate_trampoline ():
        Delegate: 0x3d7a8
        Thread: 1076754160
        AppDomain: 0
        Methode: MyResolveEventHandler
        Pre-Patch: ja
 at System.Object.__icall_wrapper_fireDelegateCtor(IntPtr )
 at App.Main() in /.../tests/assemblyresolve_event3.cs:12
[...]
Mono-RT - mono_class_init ()
        Klasse: Asm
        Thread: 1076754160
        AppDomain: 0
 at System.Reflection.Assembly.GetTypes(Boolean )
 at System.Reflection.Assembly.GetTypes()
  in /.../corlib/System.Reflection/Assembly.cs:359
 at App.Main() in /.../tests/assemblyresolve_event3.cs:17
[...]
pass.
1 test(s) passed. 0 test(s) did not pass.
```

A delegate is executed, which is resolved at run time, and the class `Asm` is loaded via Reflection at run time. So, these outputs are normal. Note, the test case assembly and all referenced assemblies have to be AOT-compiled manually in order to examine AOT-based real-time code generation.

4 Comparative Experiments

4.1 Overview

Subdirectory "comparative_experiments" contains the micro benchmarks' source codes and results. It contains one subdirectory for each evaluation platform, i.e., Linux/IA32, Linux/ARM and Windows/IA32. They include the benchmarks' sources and results (directory "Benchmarks") for each platform, as well as additional libraries (directory "Libraries").

4.2 Preparation

Linux IA32 system On Linux IA32 systems, a kernel module provides functionality to invalidate all hardware caches. This functionality is used by a library, which in turn is used by the benchmarks in order to measure execution time and invalidate caches. The kernel module's source code is located in directory "Linux_IA32\Libraries\KernelModule". The kernel module is built by executing the enclosed Makefile, which generates the kernel module `testdev2.ko`. The kernel module registers itself to a character device with major number 241. So, it has to be ensured that there is a character device file with major number 241 and name `testdev2` in the file system. This device file is created by the following command (requires super user rights):

```
$ mknod /dev/testdev2 c 241 0
```

The device file has to be created before running a benchmark. The kernel module is inserted into the Linux kernel by issuing:

```
$ insmod testdev2.ko
```

The kernel module has to be inserted before running a benchmark. The directory "Linux_IA32\Libraries\TimerLibrary" contains the source code of a library, which is used by the benchmark code in order to measure execution times and invalidate caches. It is build by executing the enclosed Makefile, which generates the shared object file `libtimer.so`. This library is needed especially for the benchmarks run with MonoRT.

Linux ARM system On the Linux ARM system used for the evaluation of MonoRT, a kernel module provides functionality to invalidate the CPU's data and instruction cache. The kernel module's source code is located in directory "Linux_ARM\KernelModule". It is built by executing the enclosed Makefile, which generates the kernel module `testdev2.ko`. It has to be ensured that the associated device file is created, see section 4.2. The directory "Linux_ARM\Libraries\TimerLibrary" contains the source code of a library, which is used by the benchmark code in order to measure execution times and to invalidate caches. It is built by executing the enclosed Makefile, which generates the shared object file `libtimer.so`.

It is necessary to enable the time stamp counter read function in order to measure execution times. This is realized by a second Linux kernel module. The directory "Linux_ARM\Librarie\TSC" contains its source code. The kernel module `rdtsc.ko` is built by executing the enclosed Makefile. It is inserted into the Linux kernel by issuing:

```
$ insmod rdtsc.ko
```

4.3 Build and Run Comparative Experiments

IBM WebSphere Real Time V2 for RT Linux The installation of IBM WebSphere Real Time V2 for RT Linux is described by [3]. The directory "Linux_IA32\Benchmarks\IBM_WebSphere" includes several subdirectories. Each subdirectory is associated with one of the comparative experiments, which are described in section 7.2 of the thesis. Each subdirectory, e.g., "Instance_Methods", which is associated with the instance methods benchmark of section 7.2.2, contains the Java source code of the benchmark, the source code of the timer library and the manual "HowTo.txt". The latter includes all steps that are necessary to build and run the micro benchmark. Each Java version of the benchmarks has its own timer library, because the Java Native Interface (JNI), which is used to call native code from Java applications, prescribes the function names of the libraries. The file "results.txt" provides the benchmark results (AOT mode).

JamaicaVM 6.0 Release 3 Information about the JamaicaVM and an evaluation version can be obtained from [4]. The directory "Linux_IA32\Benchmarks\JamaicaVM" has a has a couple of subdirectories, whose structure is similar to that of IBM WebSphere. Each subdirectory is associated with one comparative experiment. It includes the source code of the benchmark, build and run instructions ("HowTo.txt") and the results. [4] also provides references to the JamaicaVM manual, so that the build options used can be reviewd.

MonoRT The directory "Linux_IA32\Benchmarks\MonoRT" and its subdirectories include the source code, the build and run instructions ("HowTo.txt") as well as the results of the benchmarks, which are run with MonoRT on Linux/IA32. Each experiment of MonoRT has several result files. The files have self-explaining names. However, the file "results_AOT_PrePatch_reduced.txt" in subdirectory "Instance_Methods" stores the execution and startup times of the benchmark run in AOT-based Pre-Patch mode on a reduced set of CIL code (application and core library). The directory "Linux_ARM\Benchmarks\MonoRT" and its subdirectories include the source code, the build and run instructions ("HowTo.txt") as well as the results of the benchmarks, which are run with MonoRT on Linux/ARM. Each experiment of MonoRT has one result file in most cases, which stores the benchmark results in Mono's Full-AOT mode. The Linux/ARM platform is used to evaluate a checkpoint/restore approach for startup time reduction. So, the instance methods benchmark includes several result files. The file "results_PreCompilation_reduced.txt" in subdirectory "Instance_Methods" stores the execution and startup times in Pre-Compilation mode on a reduced set of CIL code (application and core library). The file "results_PreCompilation_restored.txt" in the same directory stores the execution and startup times of a checkpointed and restored benchmark process.

Reference Benchmark The directory "Linux_IA32\Benchmarks\Reference_Benchmark" includes the source code, the build and run instructions ("HowTo.txt") as well as the results of the reference benchmark written in C++. The executables are built by execution the Makefile.

.NET The directory "Windows_IA32\Benchmarks\NET" includes the source code, the build and run instructions ("HowTo.txt") as well as the results of the benchmarks, which are run with .NET on Windows/IA32.

References

1. Xamarin: Compiling Mono From Tarball. `www.mono-project.com/Compiling_Mono_From_Tarball`, last retrieved 25.05.2014.

2. Xamarin: Test Suite. www.mono-project.com/Test_Suite
3. IBM: IBM WebSphere Real Time for Real Time Linux, version 2 Information Center publib.boulder.ibm.com/infocenter/realtime/v2r0/, last retrieved 30.05.2014
4. Aicas: JamaicaVM. www.aicas.com/cms/en/JamaicaVM, last retrieved 31.05.2014