Reducing Startup Time of a Deterministic Virtualizing Runtime Environment

Martin Däumler       Matthias Werner

Operating Systems Group
Department of Computer Science
Chemnitz University of Technology

20 June 2013
Contents

1. Problem

2. Optimized Code Generation

3. Startup Time

4. Experiments

5. Summary & Outlook
Motivation

Features of modern general-purpose programming languages like Java and C#:

- Object-oriented programming
- Type safety
- Event-based programming
- Handling of Exceptions
- Dynamic class loading
- Deployment via platform-independent intermediate code
Platform Independence through Virtualizing Runtime Environment

**Figure 1:** Operating principle of a virtualizing runtime environment
Micro Benchmark

Pseudo Code:

```c
main() {
    // Initialization
    method0();
    ...
    method999();
    }

    measurement 1

    for (i = 0; i < 3; i++) {
        method0();
        ...
        method999();
        }
    }
```

Testing Environment:

- **CPU**: Intel Atom Z510@1.1 GHz
- **1.0 GiB main memory**
- **Operating System**: Linux, Kernel with RT-Preempt-Patches (Version 2.6.33.5-rt23-v1)
Results with ”Just-in-Time” (JIT)-compiler

Figure 2 : Execution times of 1000 instance methods, Mono 2.6.1, JIT mode
Results with "Ahead-of-Time" (AOT)-compiler

Figure 3: Execution times of 1000 instance methods, Mono 2.6.1, AOT mode
Results with JIT-based Pre-Compilation

Figure 4: Execution times of 1000 instance methods, Mono 2.6.1, JIT-based pre-compilation
Indirect References

programm.cs
Main () {
    ... method1();
    ...
} method1 () {
    ...
}

programm.exe
_Main :
    ... call _method1
    ...
_method1 :

Main-Methode (a)
_main :
    ... call 0x123

Main-Methode (b)
_main :
    ... call addrMethod1

Specific Trampoline @ 0x123
push method1Desc
jmp genericT ramp

Generic Trampoline
...
push method1Desc
...
call MagicT ramp
...
add 0x4, esp
ret

Magic Trampoline (.., arg, ..) {
    ...
    m = arg;
    // m : method1Desc
    ...
    addr = compile (m);
    // addr : addrMethod1
    ...
    mono_arch_patch_callsite (addr);
    ...
    return;
}

method1 @ addrMethod1
    ...
ret

C#-Code  CIL-Code nativer Code Mono-VM Code

Figure 5 : First execution of a JIT compiled method, standard Mono VM
Indirect References

1. Compilation of source code into intermediate code
2. JIT compilation of entry method at application start
3. Execution of a method-specific Trampoline
4. Execution of a more generic Trampoline
5. Execution of a top-level Trampoline
6. JIT compilation of the callee
7. Modification of method call in calling code so that it points directly to the callee
8. Trampoline return
9. Execution of the callee
10. Return to the calling code (main method)
Elimination of Indirect References in Native Code

- Logging of each emitted indirect reference ("Patch") during pre-compilation

- Execution of Trampolines before application start:
  - Modified Trampoline: return to the caller at step 9 in Figure 5
  - Redirect Patches to modified Trampoline
  - Trigger execution of indirect reference manually (step 3 in Figure 5)
  - Execution of the Trampoline mechanism (steps 4 to 8 in Figure 5)

- Utilize existing functionality to transform the native code so that it does not contain indirect references ("Pre-Patch")
Results with JIT-based pre-comp. & Pre-Patch

Figure 6: Execution times of 1000 instance methods, Mono 2.6.1, JIT-based pre-compilation and Pre-Patch
Figure 7: Startup times of 1000 instance methods, Mono 2.6.1
Reducing Startup Time

- Pre-compilation makes major contribution to startup, see Figure 7

- Reduce amount of JIT-compiled code by incorporating Ahead-of-Time-compilation:
  - Extra deployment step: compile class library and application assembly offline and store native code persistently
  - Pre-compilation: load native code from image instead of JIT-compilation
  - Pre-Patch: process Trampolines in "Procedure Linkage Table" (PLT)
Overview

Figure 8: Interplay of pre-compilation and Pre-Patch
Results with AOT-based pre-comp. & Pre-Patch

Figure 9: Execution times of 1000 instance methods, Mono 2.6.1, AOT-based pre-compilation and Pre-Patch
Startup times

Figure 10: Startup times of 1000 instance methods, Mono 2.6.1
Comparison to other solutions: overview

- **Mono 2.10.8.1 (CLI-VM):**
  - "Full-AOT": AOT-compilation of internal helper functions
  - no real-time solution

- **Microsoft .NET CLR v2.0.50727 (CLI-VM):**
  - AOT-compiler "Ngen.exe": loads native code at runtime
  - no real-time solution

- **IBM WebSphere Real Time V2 for RT Linux build 2.4 (JVM):**
  - AOT-compiler: loads native code at runtime
  - compliant to the "‘Real-Time Specification for Java’” (RTSJ)

- **JamaicaVM 6.0 Release 3 build 6928: (JVM)**
  - AOT-compiler: generates self-contained executable
  - compliant to RTSJ
Comparison to other solutions: 1000 instance methods

<table>
<thead>
<tr>
<th>System</th>
<th>Execution Time (Mean) [µs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono 2.6.1 (AOT Pre-Patch)</td>
<td>143,5275, 148,7635, 149,0615, 148,6865</td>
</tr>
<tr>
<td>Mono 2.10.8.1 (FullAOT)</td>
<td>201,13, 200,5695, 18689,747, 22289,525</td>
</tr>
<tr>
<td>IBM WebSphere RT V2 (AOT)</td>
<td>449,7015, 449,7465, 449,292, 449,67, 226,3930935, 182,5864244, 125,2151045, 125,1530462</td>
</tr>
<tr>
<td>.NET CLR v2.0.50727 (AOT)</td>
<td>1983,385, 1958,338, 1937,9825, 29207,2205</td>
</tr>
</tbody>
</table>

**Figure 11**: Mean execution time of 1000 instance methods
Comparison to other solutions: 1000 instance methods

Figure 12: Execution time span of 1000 instance methods

Experiments
Comparison to other solutions: 1000 instance methods

Figure 13 : Startup times of 1000 instance methods
Comparison to other solutions: 1000 interface methods

![Comparison to other solutions: 1000 interface methods](chart)

**Figure 14**: Mean execution time of 1000 interface methods
Comparison to other solutions: 1000 interface methods

Figure 15: Execution time span of 1000 interface methods
Comparison to other solutions: 1000 interface methods

Figure 16: Startup times of 1000 interface methods
Summary

We presented the concept of a deterministic virtualizing runtime environment.

- Pre-compilation generates all potentially executed code in advance.

- Pre-Patch eliminates indirect references in the native code.

- Code generator and code loader are not modified, so that dynamic features and legacy code are still usable.

- "Proof-of-concept" through CLI-VM Mono

- Reduction of startup time by use of an offline code generator by round 20%.
Outlook

- Further reduction of startup time
- Consideration of generic programming
- Standard compliance: ECMA-335 §8.9.5 does not allow execution of Type constructors before first reference.
End of the presentation.

Thank you for your attention.
Feel free to ask.