RT.NET – Towards a Real-Time Implementation of the ECMA Common Language Infrastructure

Distributed Real-time Systems

Norman Kluge
Agenda

- Introduction

- Real-Time Aspects
  - Just-In-Time Compiler
  - Memory Management
  - Hardware-near programming
  - Other Features / Non-Features

- Summary
Common Language Infrastructure (CLI)

- Standardized
  - ISO / IEC 23271
  - ECMA-335
- Developed driven by Microsoft
- Parts:
  - Common Type System (CTS)
  - Common Intermediate Language (CIL)
  - Virtual Execution System (VES)
  - Standard Library
- Implementations:
  - .NET, .NET Compact Framework
  - Mono
About RT.NET

- Approach to integrate Real-Time capabilities into CIL

- Consists of
  - GCC IL Front-End
  - Run-Time support library
  - CLI Standard Library
  - Target-specific interfaces

- Developed in 2006

- By Martin von Löwis and Andreas Rasche

- Used in the Higher Striker experiment
Why RT.NET?

- CLI is not Real-Time capable
- No prediction of execution times
- No implementations of CLI for most of embedded platforms
- Need of Hardware-near programming
- Real-Time specifications
Agenda

- Introduction
- Real-Time Aspects
  - Just-In-Time Compiler
  - Memory Management
  - Hardware-near programming
  - Other Features / Non-Features
- Summary
No implementation for most of platforms used in embedded systems

Causes hard to predict execution times

Causes overhead for compilation

Compile code before runtime
Developed by Jan Möller

Similar approach to gcj compiler

Translates CIL bytecode into Register Transfer Language (RTL)

GCC is used to translate RTL into target specific architecture / operating system binaries
Agenda

■ Introduction

■ Real-Time Aspects
  □ Just-In-Time Compiler
  □ Memory Management
  □ Hardware-near programming
  □ Other Features / Non-Features

■ Summary
Worst-case execution time depends on
- Allocation itself
- Initialization

Initialization
- Memory has to be zero-initialized (per-class constant)
- Constructor invocation-time is high predictable (non virtual)

Allocation
- $O(1)$
Fixed-size MM with Segregated Free Lists

- memory pools with fixed size blocks
- block sizes and number of blocks per size statically assigned
- no external fragmentation, internal fragmentation depends
- $O(1)$

- only useful for applications with well-known memory needs
  - networking code, protocol stacks

[Matthias Richly, 2010]
Object Allocation 3/3

- Typically multiples of 8 byte blocks
- Per size class a linked list
- On memory request, the next higher size class is chosen
- The first block of the per size class list is taken
- Additionally a global pointer of free space
  - If block-specific list is empty, a block from this list is taken
- Deallocation returns the block into the front of the list
- Synchronized operations
Garbage Collector

- Garbage Collector causes hard to predict execution times
  - Could start at any point of application
  - Could freeze execution for a long time

.enabled GC and use conventional methods
Reference Counting 1/2

- Each object has a reference counter
- Atomic operations are used (hopefully)
- $O(1)$

- If reference counter is zero, object memory may deallocated
- De allocation could result further deallocations
- $O(n)$
As consequence of $O(n)$, the system is not fully predictable
- Old objects could be queued and may later dereferenced by a low priority thread

Cyclic data structures
- Application developer should manually break these cycles

Finalization [not implemented]
- Invoke finalization method immediately (hard to predict)
- Use a queue (see above)
Agenda

■ Introduction

■ Real-Time Aspects
  □ Just-In-Time Compiler
  □ Memory Management
  □ Hardware-near programming
  □ Other Features / Non-Features

■ Summary
Embedded systems often deal with special hardware

Application may directly access the hardware

Attributes (annotations) for marking fields for direct I/O address mapping

- [MemoryAlias(addr)] for memory mapped I/O
- [PortAlias(addr)] for port based I/O

[Polze, Rasche, Rabe, 2007]
public struct Port
{
    [MemoryAlias(0x00)]
    public byte DataDirectionRegister;
    [MemoryAlias(0x02)]
    public byte DataRegister;
}

public class H8_3297
{
    /* ... */
    [MemoryAlias(0xFFB5)]
    public static Port Port4;
    [MemoryAlias(0xFFB8)]
    public static Port Port5;
    /* ... */
}

byte b = H8_3297.Port5.DataRegister;
H8_3297.Port5.DataRegister = ~0x42;
H8_3297.Port4.DataDirectionRegister |= 0x23;

[Polze, Rasche, Rabe, 2007]
Memory mapped / Port based I/O 3/3

- Only one mapping per field
- Fields may only be a closed value type
- Address may not used by the runtime (heap / stack)
- None or all fields of a specific type may be tagged
- All fields may tagged with MemoryAlias or all with PortAlias
- Address computation
  - Field is static $\rightarrow$ Address is the attached one
  - Otherwise $\rightarrow$ Address is the sum of the attached one + the address of the parent object
Interrupts

- **Interrupts are represented by delegates (function pointers)**

```csharp
[InterruptHandler]
void delegate aHandler(void);

[InterruptHandler] static void IcqOH() {
    /*...*/
    public static void Main (){

        H8_3297.VectorTable.NonMaskableInterrupt = IcqOH;

    }
    // OEM Library
    public class H8_3297
    {
        [MemoryAlias(0x0000)]
        public static VectorTable VectorTable;
        /* ... */
    }
```

[Polze, Rasche, Rabe, 2007]
Scheduling

- Scheduling features of underlying Real-Time OS are used
- Accessing Thread objects (especially CurrentThread) should be O(1)
  - Thread-local storage is used
- System.Threading distinguishes only five priorities
  -Enumerating could be extended
- New PeriodicThread class
- Mutex extended for the priority ceiling protocol
- Extensions follow Posix 1003.1b (real-time) and Real-Time Specification for Java (RTSJ)
Agenda

- Introduction
- Real-Time Aspects
  - Just-In-Time Compiler
  - Memory Management
  - Hardware-near programming
  - Other Features / Non-Features
- Summary
Other Features / Non-Features

- **Linker**
  - Memory on embedded systems is often restricted
  - Only code which is really needed is linked

- **Exception-handling**
  - Exception-handling causes hard to predict execution times
  - Not implemented → Exception causes program termination
  - Check whether an exceptions occurs before it actually does
    - Application developer may be aware of this
Agenda

- Introduction
- Real-Time Aspects
  - Just-In-Time Compiler
  - Memory Management
  - Hardware-near programming
  - Other Features / Non-Features
- Summary
Conclusions

- Approach for real-time applications based on CLI
- Targeting different CPU’s on different operating systems
- For basic operations a WCET can predicted
- Target-specific interfaces for Mindstorm™ RCX on brickOS
Future Work

- Implement all opcodes
- Support for driver development
- Support for dynamic reconfiguration (Adapt.NET)
- Integrating of whole-program analysis tool to reduce the set of library functions required to run a certain application
- Implement RTOS scheduling features (like Ada)
References

- Martin v. Löwis, Andreas Rasche: Towards a Real-Time Implementation of the ECMA Common Language Infrastructure
- Andreas Polze, Andreas Rasche, Bernhard Rabe: 5. Real-Time Programming. In: Embedded Operating Systems, WT07/08
- Martin v. Löwis, Jan Möller: A Microsoft .NET Front-End for GC