Towards a Real-Time Implementation of the ECMA Common Language Infrastructure

Martin von Löwis and Andreas Rasche
Hasso-Plattner-Institute at
University of Potsdam, Germany

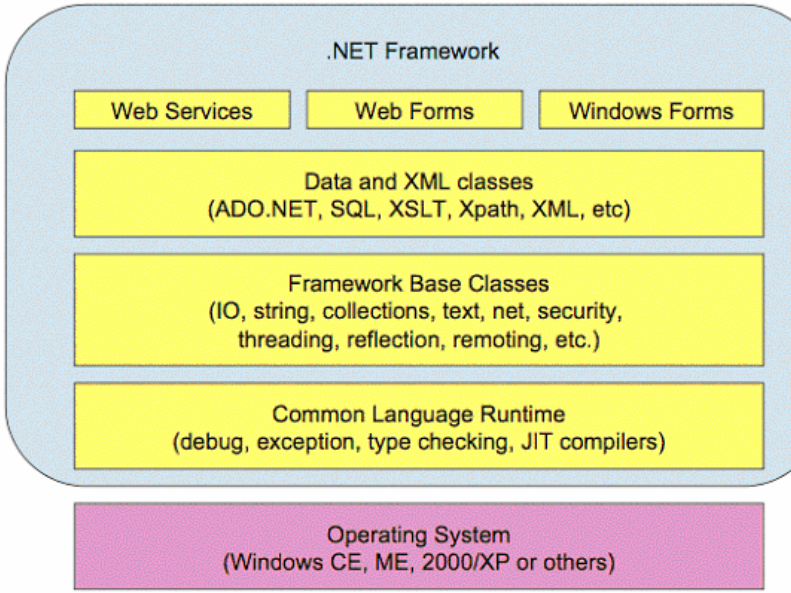
ISORC 2006
24-26 April, 2006
Gyeongju, Korea

Outline

- Motivation
- GCC Common Intermediate Language (CIL) Front-end
- Prediction of Execution Times
- Extending the Base Class Library
- Real-Time.Net on Windows CE.NET
- The Higher Striker Experiment
- Experimental Evaluation
- Conclusions

The Common Language Infrastructure

- Open specification developed by Microsoft
  - Describes executable code
  - Runtime environment
  - Describes core of the Microsoft .NET Framework
- Standardized by ISO/ECMA:
  - ISO/IEC 23271
  - Standard ECMA-335



- Common Language Runtime (CLR) is Microsoft's commercial implementation of the CLI
- Open source implementations: Mono, Portable.Net

C# and CLI for Real-Time Systems

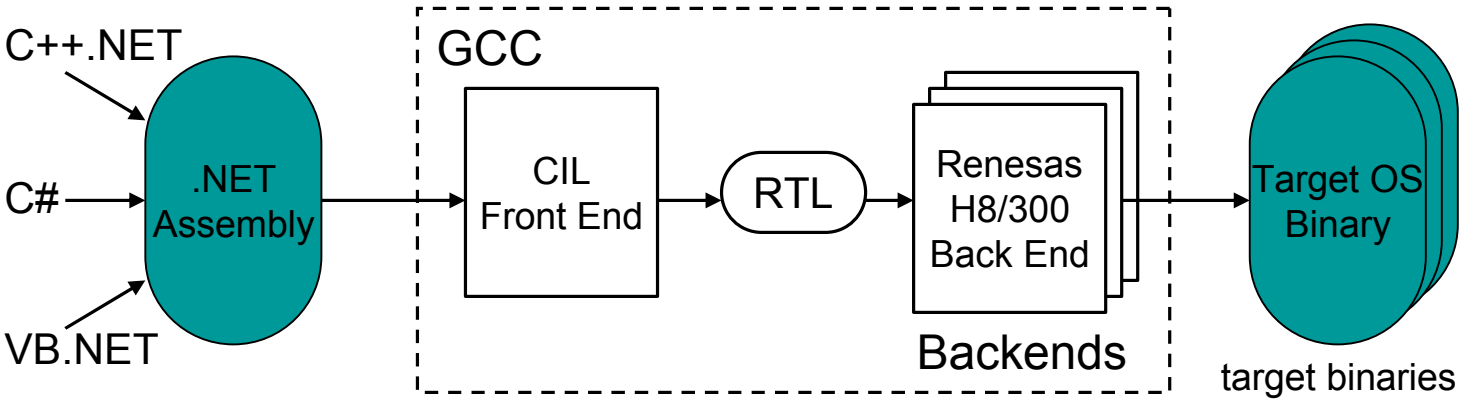
- Higher-level programming languages induce higher developer productivity
  - Programming errors never result in system crashes because of predictable nature of sandbox-mode execution of CLI instructions
  - due to the support for rapid prototyping, simulators for targets can be created more easily.
  - Usage standard library functions of the CLI minimizes code for target-specific hardware.
- Find a way to run CLI intermediate language (IL) code on the target
- Find a way to integrate the CLI program with the “special” hardware of the target
- Find a way to provide real-time guarantees to developers of applications for the target

ECMA CLI for Real-Time Systems

- **Just-in-time compilation** causes hard-to-predict execution times for method execution
- **Garbage collection** freezes application code for potentially long time
- **Memory allocation actions**, such as object creation, vary depending on whether garbage collection is invoked to free unused memory for reuse
- No direct hardware access - **I/O-programming** (interrupts, direct memory access ...)
- Limited **threading** model (semantics, priorities, synchronization)

GCC Intermediate Language Front-End

- Modification the GNU Compiler Collection (GCC) to support compilation of IL code into target machine code
  - Compilation of each method in intermediate language into one function on assembler level
  - Symbolic execution to translate the stack machine instructions into GCC's internal statement representation (GNU compiler for Java (GCJ) approach)
  - GCC optimizes and compiles into assembler code



Integrating with the Target

- Implementation of specific IL opcodes use run-time support library
  - Primarily memory management and exception handling (e.g. newobj run-time function)
- Implementation of the standard CLI library
  - No support planned for reflection and security
  - Library extensions for real-time systems
- Implementation of target-specific interfaces
  - Mindstorms™ RCX: brickOS API exposed as predefined class brickOS (with classes such as dmotor, dsensor, or dsound)

Predictability of execution times (WCET)

- Hardware-supported constant time execution
  - loading constants, access to local variables, arithmetic operations, method calls …
- Memory Management Operations
  - newobj, newarr, box stloc.*, stelem.*, starg.*, stind.*, stfld
  - Zero-initialization: linear to object/array size
  - Pool memory allocator: constant time (de-)/allocation
  - Reference counting: object deallocation linear with the number of objects that become unreferenced
- Exception Handling: hard to predict
- Library functions: WCET based on IL-level and RTOS characteristics

Multi-threading and Scheduling

- Usage of underlying RTOS scheduling features
- Extended control over thread-priorities
  - Access to all RTOS priorities
- New thread classes for periodic thread support
- Enhanced synchronization mechanisms (priority inheritance, priority ceiling)
- Integration of scheduling algorithms by implementing new Scheduler classes
- Enhanced precision of .Net Timer classes
- Extensions follow Posix 1003.1b (real-time) and Real-Time Specification for Java (RTSJ)

ExtendedMutex class

```
enum MutexProtocol {PriorityCeiling,PriorityInheritance,None};

class ExtendedMutex: Mutex
{
    public ExtendedMutex();
    public ExtendedMutex(PriorityCeiling ceiling);
    ...
    public bool WaitOne(int millisecondsTimeout,bool exitContext);
    public void ReleaseMutex();
    public void Close();
    public ThreadPriority PriorityCeiling{get{};set{};}
    public MutexProtocol Protocol{get{};set{}}
    ...
}
```

Periodic Thread Class

```
public delegate void ThreadStart();
public class PeriodicThread
{
    PeriodicThread(ThreadStart start, RelativeTime period);
    static void WaitforNextPeriod();
    static void Sleep(RelativeTime time);
    static PeriodicThread CurrentThread();
    void Start();
    ThreadPriority Priority {get{} set{}};
    ...
}
```

I/O Subsystem in Real-Time.Net

- .NET Attributes for marking fields for direct I/O address mapping
  - `[MemoryAlias(addr)]` for memory mapped IO
  - `[PortAlias(addr)]` for port based IO

```
struct Port {
    [MemoryAlias(0xffb2)] static byte DataRegister;}
```

- Hardware vendor code provides class implementing their hardware
  - Implemented as `struct` with field-annotations
  - `H8_3297.Port1.DataRegister = 0xFF`

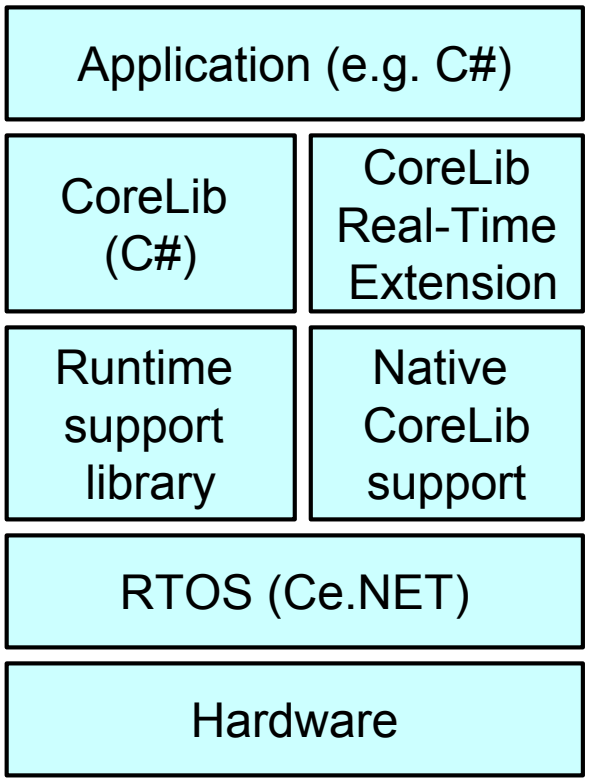
I/O Subsystem: Interrupts

- Interrupt vectors are `delegate` fields of type
  `void delegate InterruptHandler (void)`
- Interrupt handlers are `static` methods with `[InterruptHandler]` attribute
  - to make GCC generate different return opcode “return from interrupt”

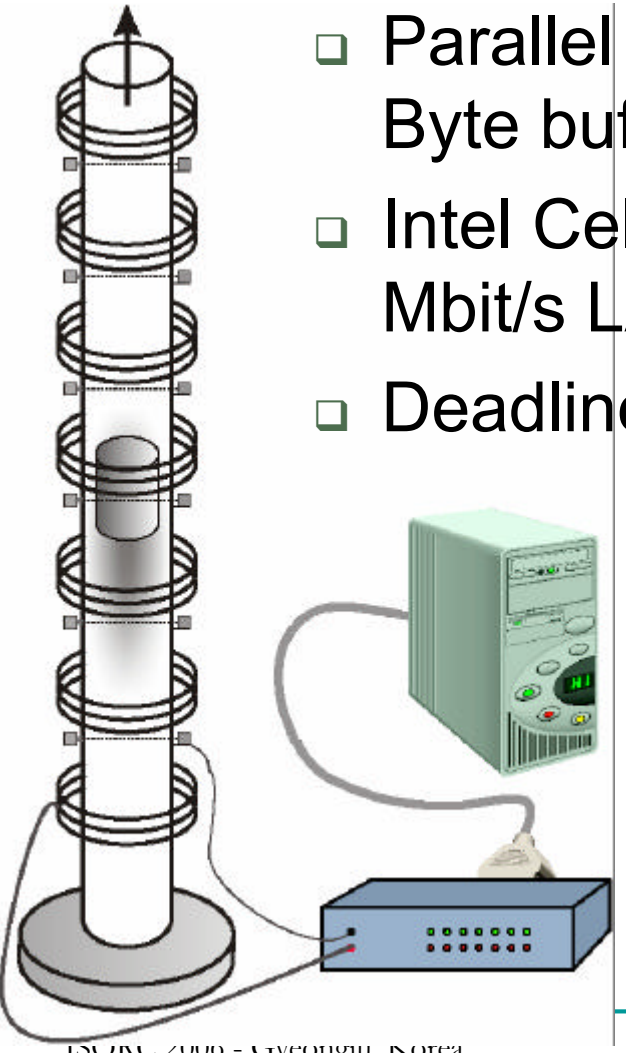
```
[InterruptHandler] static void Irq0H() {
 /*...*/
}
H8_3297.Irq0 = Irq0H;
```

Real-Time.Net on Windows CE.NET

- Implementation of runtime support library and base class library (mscorlib.dll) using CE.NET API
- Runtime support library in C
- Most parts of mscorlib.dll in C#
- Parts must be implemented natively
  - CurrentThread uses thread local storage
  - Mapping to RTOS API
  - String functions, Printing to screen ...



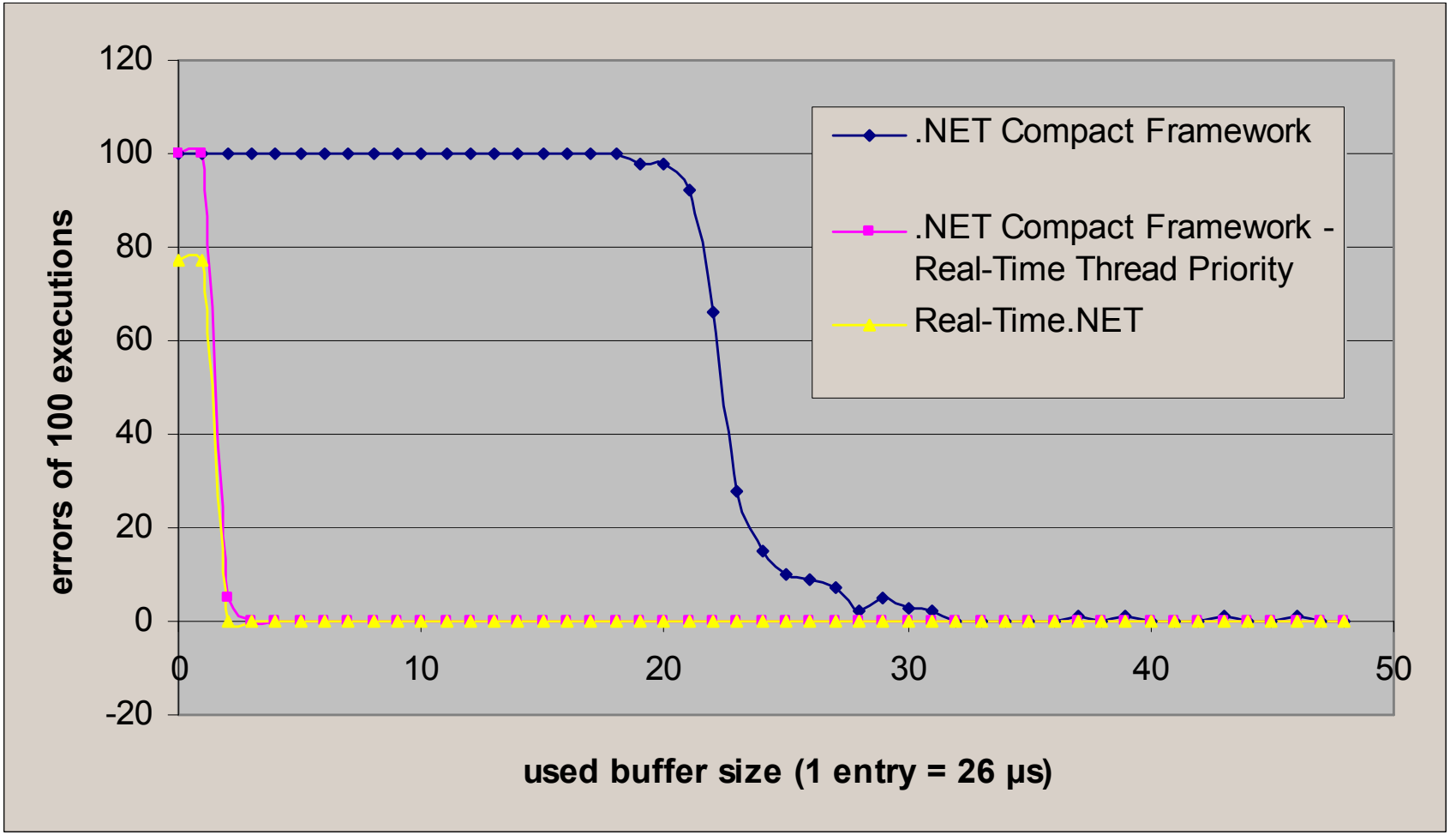
Real-Time.Net in Action
The Higher Striker Experiment



- Parallel I/O / 38,4 kHz sample rate / 256 Byte buffer (configurable 0-256)
- Intel Celeron 633 MHz, 128 MB RAM 10 Mbit/s LAN (NE 2000 PCI)
- Deadlines / reaction times depend on used buffer size

```
INITIALIZE(buffer_size);
while(true)
{
    READ();
    WRITE(buffer);
    while(GETSTATE()==EMPTY_LS);
}
```

Maximum Jitter Measurements



Current Status & Ongoing Work

- Support for driver development (including interrupt handling and hardware I/O) in C#
- Integration of whole-program analysis tools to reduce the set of library functions required to run a certain application
- RTOS independent implementation
  - Implementation of RTOS features in C#
- Support for dynamic reconfiguration of systems, based on the Adapt.NET framework

Conclusions

- Real-Time.Net : developing real-time applications based on the ECMA Common Language Infrastructure
- Predictable Execution of C# code
- Efficient code execution: compilation of intermediate code to native machine instruction
- Introduction of new functions required for real-time application development (extended base class library)
- Radip application development for real-time applications

Towards a Real-Time Implementation of the ECMA Common Language Infrastructure

Martin von Löwis and Andreas Rasche
Hasso-Plattner-Institute at
University of Potsdam, Germany

ISORC 2006
24-26 April, 2006
Gyeongju, Korea