Hardware-near Programming in the Common Language Infrastructure

Stefan Richter, Andreas Rasche and Andreas Polze
Hasso-Plattner-Institute at University of Potsdam

ISORC 2007
7-9 May, 2007
Santorini Island, Greece
Roadmap

- The Common Language Infrastructure
- Real-time.Net Project
- GCC Common Intermediate Language (CIL) Front-end
- Hardware-near programming
  - Direct-memory access
  - Interrupt handling
  - Support for Implementing Concurrency
- 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 support for **hardware-near programming**
  - Direct Memory Access
  - Memory Mapped I/O
  - Interrupt Handling
  - Low-level access to registers (scheduler support)
- Limited **threading** model (semantics, priorities, policies, 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

![Diagram showing the flow of GCC Intermediate Language Front-End]

- GCC 
  - .NET Assembly
  - C++ .NET
  - C#
  - VB.NET 

- CIL Front End
- RTL
- Back End
- Renesas H8/300
- Back End
- Target OS Binary
- target binaries
I/O Subsystem in Real-Time.Net

- Mapping hardware to the CLI’s object model
- 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

```csharp
[MemoryAlias(0xff82)]
static byte myMemoryLocation;

byte b = myMemoryLocation;
myMemoryLocation = ~0x42;
myMemoryLocation |= 0x23;
```
## Structured Definition of Hardware

```csharp
public struct Port{
    [MemoryAlias(0x00)]
    public byte DataDirectionRegister;
    [MemoryAlias(0x02)]
    public byte DataRegister;
}

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

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

- Hardware vendors implement library for their hardware
- Hardware access defined by vendor-specific struct
Rules for Memory/Port Aliases I

- **Uniqueness**: At most one alias attribute per field
- **Validity**: Addresses must be valid
- **Fields the type of which is a *Closed Value Type* only**
  - CVTs are value types that contain value types only
- **Completeness**
  - All fields in a type must have the same alias type
  - Static fields must have the same alias type as their type
- **No access optimisation**
- **No memory management**
Rules for Memory/Port Aliases II

- Let $x$ be the value attached to a field $f$ by an alias attribute. If $f$ is static, its address is $x$.
  - `ldsflda f`: loads $x$ on the stack

- Otherwise, its address is the sum of $x$ and the address of the field that $f$ is a field of.
  - `ldflda f`: takes an address $a$ from the stack and pushes $a+x$
Rules for Memory/Port Aliases III

- Let $f$ be a MemoryAlias($x$) attributed field the type of which is a built-in type of size $s$. Then, the memory block of size $s$ starting at the alias address of $f$ is never to be used for memory allocation. Accesses to $f$ are to be redirected to its alias address:
  - $ldsflt$ $f$: reads from address $x$ and pushes that value on the stack
  - $stsflt$ $f$: pops a value from the stack and writes it to address $x$
  - $ldflt$ $f$: takes an address $a$ from the stack, reads from address $a+x$ and pushes that value on the stack
  - $stflt$ $f$: takes an address $a$ from the stack, then pops a value from the stack and writes it to address $a+x$

- Analogous for PortAlias
public static void Sound (ushort frequency)
{
    Off ();

    if (frequency < 31)
        return;

    H8_3297.Board.Timer8Bit.Channel0.ControlStatusRegister = 0x03;
    H8_3297.Board.Timer8Bit.Channel0.TimerCounter = 0x00;

    if (frequency <= 122)
    {
        H8_3297.Board.Timer8Bit.Channel0.ConstantRegisterA =
            (byte) ((short) 7813 / frequency);

        H8_3297.Board.SerialTimerControlRegister &= 0xFE;
        H8_3297.Board.Timer8Bit.Channel0.ControlRegister = 0xOB;
    }
    else if (frequency <= 488)
    {
        H8_3297.Board.Timer8Bit.Channel0.ConstantRegisterA =
            (byte) (31250 / frequency);

        H8_3297.Board.SerialTimerControlRegister |= 0x01;
        H8_3297.Board.Timer8Bit.Channel0.ControlRegister = 0xOB;
    }
I/O Subsystem: Interrupts

- **Interrupt vectors** are delegate fields of type `void delegate InterruptHandler (void)`

- **Interrupt handlers** are static methods with `[InterruptHandler]` attribute

- Parts of immediate working context that is manipulated must be saved on entry and restored on exit of interrupt handlers method

- Immediate interrupt context must be restored at the end of interrupt handler methods
  - Our GCC frontend generates different return opcode “return from interrupt”

- **Static functions for controlling interrupt hardware**
  - `Hardware.Cpu.Interrupts.DisableAll`
  - `Hardware.Cpu.Interrupts.EnableAll`
Interrupt Handling Example (manufacturer)

[StaticDelegate]
public delegate void InterruptHandler();

public struct VectorTable
{
    [MemoryAlias(0x06)]
    public InterruptHandler NonMaskableInterrupt;
}

public class Hardware
{
    [MemoryAlias(0x0000)]
    public static VectorTable VectorTable;
    [MemoryAlias(0xFFC3)]
    public static byte SomeRegister;
}
Interrupt Handling Example (programmer)

```c
[InterruptHandler]
static void Handler ()
{
    /* any code */
}

public static void Main ()
{
    Hardware.VectorTable.NonMaskableInterrupt = Handler;
}

or in C# 1.0:

Hardware.VectorTable.NonMaskableInterrupt =
    new InterruptHandler (null, Handler);
```
Implementing Preemptive Concurrency

static void Main ()
{
    Cpu.Scheduler = DetermineNext;
    Hardware.VT.TimerInterruptVector = Cpu.InvokeScheduler;
    Timer.Start ();
    while (true) Cpu.Sleep ();
}

// Class Cpu is implemented target specific
public static Cpu.Context DetermineNext (Cpu.Context context)
{
    Cpu.Context newContext;
    /* determine the new context */
    return newContext;
}
Status and Ongoing Work

- Our gcc front-end supports interrupt handling and direct memory access as suggested.
- Acceptable overhead (max. 50%) of generated code compared to "hand-written" assembler.
- Acceptable compilation times.
- Supported target Platforms: H8-300 Lego Mindstorm RCX 2.0.
- Current projects:
  - OS#: micro-kernel operating system using Real-Time.Net.
  - Support for latest Lego NXT hardware (ARM).
Conclusions

- New extension for the standard ECMA 335
- General approach for mapping hardware registers to structured value types
- Enables OEMs to specify their hardware in high-level languages
- Attributed-based declaration of interrupt handler methods
- Approach for hardware-near programming using more productive high-level languages
Hardware-near Programming in the Common Language Infrastructure

http://www.dcl.hpi.uni-potsdam.de

Stefan Richter, Andreas Rasche and Andreas Polze
Hasso-Plattner-Institute at University of Potsdam

ISORC 2007
7-9 May, 2007
Santorini, Greece