Hardware-near Programming in the Common Language Infrastructure

Stefan Richter, Andreas Rasche and Andreas Polze

Hasso-Plattner-Institute at University of Potsdam

14482 Potsdam, Germany

stefan.richter|andreas.rasche|andreas.polze@hpi.uni-potsdam.de

Abstract

Virtual machine-based programming languages, such as Java and $C^{\#}$ have made the programming of desktop computer systems simpler, less error-prone and more efficient. Embedded systems development rarely benefits from this advantages. This is because the disciplines special needs, such as direct hardware access and timelininess, are rarely considered in in these environments. In particular, virtual machines usually do not allow for accessing hardware directly, making it impossible to express substantial parts of embedded systems inside the virtual environment. By specifying additional rules, describing an implementation of a conforming compiler, and presenting examples, we show how the virtual machine defined by the ECMA standard 335 can be carefully extended to support hardware-near programming.

1 Introduction

The Common Language Infrastructure as defined by the standard ECMA 335 is a virtual machine designed as a target for compilers of many different programming languages [3]. It is fully object oriented, ie. the only way of allocating memory is by creating instances of objects. For that purpose, it comprises an object model in the tradition of languages like C++ and JAVA: less expressive than the former but more powerful than the latter. Every location (variables, stack locations, method arguments, etc) is typed. Types can be reference types or value types; while an instance of a reference type does always point to another value - eg. interfaces and classes are reference types -, a value type describes structured pieces of memory (comparable to a struct in C). Furthermore, it is important to know that the CLI allows for attaching meta-data annotations to most program items. We use these meta-data annotations, called attributes, for extending the CLI.

When developing embedded systems or operating systems, programmers often have to write hardware-near code [13]. At present, no provisions have been made in the standard ECMA 335 that allow programmers to write such code
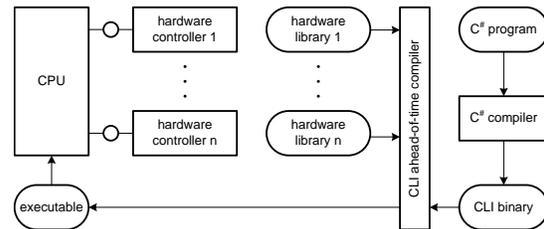

Figure 1. Compiler and hardware libraries

from inside the CLI. While it is in general possible to use external methods written in platform-specific assembly language, C or similar languages for this purpose, such a solution is rather dissatisfying because it lacks the safety of the CLI code and adds portability issues.

In order to show how programmers can be enabled to write hardware-near code from inside the CLI, we developed extensions for three important issues that arise in the course of embedded systems development:

1. Direct hardware access - reading and writing of memory-mapped and port-based hardware registers

2. Interrupt handling - declaring interrupt handlers and registering them with the CPU's interrupt handling sequence

3. Concurrency - implementing arbitrary task models (such as thread schedulers) and entering the CPU's power-saving mode

In an ideal course of action in embedded software development, we favor a strong division of labor: Every piece of hardware shall be described by a special CLI library produced by the hardware manufacturer. With that, embedded systems programmers need not know about locations of hardware registers or special access to them. Instead they can program embedded systems as they would do with ordinary systems, by using identifiers.

One major goal of our work was to reduce the amount of assembler and other non-CLI code. Whenever feasible, we prefer to enhance a compiler by safe means of expression instead of allowing the execution of arbitrary code. In the

course of the *Real-Time.Net* [12] project at Hasso-Plattner-Institute, a compiler for translating CLI intermediate instructions to target machine code, was built by developing a new front end for the GNU Compiler Collection (GCC) [5][11].

In this paper, we are going to present the extensions we developed by giving formal definitions and rules, that specify additions to those in ECMA 335, as well as examples. After that, the account is complemented by some remarks on implementation and performance of our prototypical compiler. Finally, we give an overview on alternative approaches, both existing and possible ones.

2 Extensions

2.1 Direct Hardware Access

In principal, each piece of hardware is accessed by its registers that are connected to a bus such that they can be written to and read from by the CPU. Depending on the system architecture, there are two ways of accessing these registers: memory-mapped I/O and port-based I/O. In the former case, hardware registers share the memory's address space and are accessed by the very same CPU instructions as the memory cells; in the latter case, a separate address space with special read/write CPU instructions is used. In fact, memory can be considered a hardware device as well.

There is a common abstraction for memory cells in most programming languages: variables. Hence, it suggests itself to extend the concept of variables for use with arbitrary hardware registers. For this purpose, we propose the usage of two new attribute classes `MemoryAliasAttribute` for memory-mapped I/O and `PortAliasAttribute` for port-based I/O along with a couple of rules in addition to those of ECMA 335. We call those attributes *alias attributes*.

2.1.1 Exemplified Application

Suppose, we want to develop a CLI library for the RENESAS H8/3297 board [9]. On that board with a 16-bit address space, we can find a *port 4*[1] with a *data direction register* and a *data register* that are accessible at addresses 0xFFB5 and 0xFFB7, resp. For that purpose, we would have created a C# source file containing the following class fragment:

```
public class H8_3297
{
  /* ... */
  [MemoryAlias(0xFFB5)]
  public static byte Port4DataDirectionRegister;

  [MemoryAlias(0xFFB7)]
  public static byte Port4DataRegister;
  /* ... */
}
```

---

[1]In this context, the term *port* denotes something different than in *port-based I/O*.

Both fields do not correspond to memory cells but to the registers. Nevertheless, an application programmer can access them like ordinary variables, eg:

```
byte b = H8_3297.Port4DataRegister;
H8_3297.Port4DataRegister = ~0x42;
H8_3297.Port4DataDirectionRegister |= 0x23;
```

Still, this example is not very well structured, especially when port 5 and 6 of the H8/3297 series are known to have the very same registers as port 4, with the addresses laid out analogously. Thus, we had better used a value type `Port` and changed the definition of class `H8_3297`:

```
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;
  /* ... */
}
```

Now, we have the *static* fields `Port4` and `Port5` the type of which is `Port`. This value type comprises two *instance* fields, the addresses of which are *relative* to the addresses of `Port4` and `Port5`. Like above, all these fields do not correspond to memory locations but to registers and can be used as usual:

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

Of course, the type of an alias-attributed instance field could be a value type containing alias-attributed instance fields itself, allowing for an arbitrary modularization of registers.

2.1.2 Formal Definition

For ease of language, we introduce a few definitions.

Definition 1 *A closed value type is a value type all instance fields of which have closed value types. Built-in value types are closed value types.*

Hence, a closed value type is a value type that contains no references to objects, etc.

Definition 2 *A component of a closed value type `T` is either an instance field of `T` or a component of the type of an instance field of `T`. A component that has a built-in value type is basic.*

All the basic components of a value type make up the physical locations (ie memory cells or in our case hardware registers), while all other components are merely abstract entities for structuring purposes.

Obviously, we need uniqueness and validity rules:

Rule 1 *A field must not have more than one alias attribute.*

Rule 2 *The usage of an alias attribute must specify an address that exists in the respective address space on the underlying platform and that is not in use by the runtime for internal purposes such as providing for stack and heap.*

Furthermore, we do not want to allow the application of an alias attribute to a field the type of which is a class type because such a field must be assigned `null` or a pointer to an object that is subject to automatic memory management. But since these fields are not necessarily memory cells and may thus not always *contain* the pointer after the *assignment* of that pointer, there would be no clear semantics for garbage collection. Therefore, we need the restriction to closed value types.

Rule 3 *Alias attributes may only be used on instance fields of closed value types and static fields whose type is a closed value type.*

With the next rule, it is always clear, whether a given structure is an alias or an entity that is to be allocated on the heap. This is important when pushing the address of that structure on the virtual execution stack. It would be necessary to push both addresses, the address of that location for its non-alias-attributed parts and the base address for the alias attributes. This is an unnecessary overhead induced by syntactic sugar as you can always get along by having separate types for aliases and common value types.

Rule 4 *The components of a closed value type `T` must either all be attributed with a `MemoryAlias` or none of them. Accordingly, any static field that has type `T` must be attributed or must be not attributed. An analogous proposition holds for `PortAlias`.*

The next rule targets a similar problem: We forbid the declaration of a static field without an alias attribute if the type of that field contains alias attributes. Suppose, we had this value type `T`:

```
public struct T
{
  [MemoryAlias(0x00)]
  public byte x;

  [MemoryAlias(0x24)]
  public byte y;
}
```

and two static fields `a` and `b` of that type, `a` attributed properly, `b` not. Let `T` have a member function `m`. The implicit instance argument of `m` would be an address of a location of type `T`. When we pass `a` we certainly want to pass the alias address of `a` (in fact we do not have another one), in case of `b` we had to pass the usual address of `b`. But the layout of `T` is not the same for `a` and `b`[2], which alone is reason enough

_____

[2]If the layout was the same, we would waste quite a lot of memory. In the case of `a.x` and `a.y` having addresses at both ends of the memory space, this waste would be massive for `b`.

for our rule. Moreover, we would need *two* versions of `m`, one for `a` and `b` each (or one method containing `m`'s code twice). With respect to the limited resources of embedded systems we consider this unacceptable.

Rule 5 *For any closed value type `T` and any static field `f` that has type `T`, the alias attributes must correspond: If `T` contains fields with alias attributes then `f` has an alias attribute of the same type.*

For the specification of the semantics, we first detach alias attributed types from the usual memory allocation.

Rule 6 *Types that have alias attributed instance fields are never subject to the runtime's memory management.*

Next, we give a recursive rule for how addresses are obtained. Note, that because of our restrictions above, we will have a static alias attributed field for every non-static attributed field. Therefore, this recursion will always have an end, thus defining the address of every field. Further note, that the recursion is somewhat implicit because it results from the semantics and usage of `ldflda`; whenever there are nesting depths greater than one, say `x.y.z`, the addresses are loaded sequentially by first using `ldsflda` followed by an appropriate number of `ldflda`, in our example (simplified): `ldsflda x; ldflda y; ldfld z`.

Rule 7 *Let x be the value attached to a field f by an alias attribute. If f is static, its address is x. Otherwise, its address is the sum of x and the address of the field that f is a field of. In terms of instructional semantics this means:*

1. `ldsflda f` *loads x on the stack*

2. `ldflda f` *takes an address a from the stack and pushes $a + x$*

Finally, we can now give the essential rules for the direct implications of alias attributes. After all, they are straightforward without need for explication. Just note, that due to our restrictions, we can only use alias attributes in conjunction with instance or static fields. This means, we need not specify any accesses to local variables or arguments.

Rule 8 *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:*

- `ldsfld f` *reads from address x and pushes that value on the stack*

- `stsfld f` *pops a value from the stack and writes it to address x*

- `ldfld f` *takes an address a from the stack, reads from address $a + x$ and pushes that value on the stack*

- `stfld f` *takes an address a from the stack, then pops a value from the stack and writes it to address $a + x$*

Rule 9 *Let f be a* `PortAlias(x)` *attributed field the type of which is a built-in type of size s. Accesses to f are to be translated to accesses to x using the special instructions of the underlying platform for port-based I/O:*

- `ldsfld f` *reads from address x and pushes that value on the stack*

- `stsfld f` *pops a value from the stack and writes it to address x*

- `ldfld f` *takes an address a from the stack, reads from address $a + x$ and pushes that value on the stack*

- `stfld f` *takes an address a from the stack, then pops a value from the stack and writes it to address $a + x$*

2.2 Interrupt Handling

Definition 3 *The* immediate interrupt context (IIC) *is that part of the CPU that is saved automatically on the stack when an interrupt occurs. The* immediate working context (IWC) *are all other registers of that CPU.*

In general, there are seven steps to be performed, when an interrupt occurs:

1. The CPU saves the IIC on the stack.

2. The CPU calls the interrupt handler via the interrupt vector table.

3. The interrupt handler saves the IWC on the stack.

4. The interrupt handler performs its task.

5. The interrupt handler restores the original IWC.

6. The interrupt handler returns using a special return instruction (or sequence).

7. The CPU restores the original IIC.

While steps 1 and 7 are beyond the programmer's control and step 4 is the same as in any method, steps 2, 3, 5, and 6 need special care. In order to tell the compiler to generate code that implements steps 3, 5, and 6, we provide for an attribute `InterruptHandlerAttribute` that may be attached to any method.

Rule 10 *If the attribute* `InterruptHandler` *is applied to a method m, the compiler must make sure that the immediate interrupt context is restored on exit of m. Furthermore, any parts of the immediate working context that are manipulated by m must be saved on entry and restored on exit of m.*

For step 2, we must enable the programmer to put the address of a method into any of the slots of the interrupt vector table. For that reason, we first need to give the manufacturer a means to describe the interrupt vector table. Using the concept of alias attributes, they could use an attributed value type as shown above. The problem arises when looking at the type of the fields that stand for the vector table entries. Unfortunately, a delegate[3] contains not only a method pointer but also a pointer to an object, in case it points to an instance method. Being to large, delegates can therefore not be used for the interrupt vector table. On the other hand, delegates are the only safe way for using method pointers in the CLI. For that reason, we introduce a new kind of delegate (called *static delegates*). Each delegate type that is attributed `StaticDelegate` may only be used with static methods. With that, the field for the object pointer can be omitted.

Rule 11 *A delegate type* `D` *with signature S that is attributed* `StaticDelegate` *is a value type comprising a single field of type* `native int`*. Only a limited number of operations are specified for such a type:*

1. *The constructor pops two arguments from the evaluation stack, the first of type* `System.Object`*, the second a token referring to a compatible static method as defined in ECMA 335 Partition II § 14.6.1. When called, the first argument is always ignored and should be* `null`*; the second argument is put on the stack (converted to* `D` *if the compiler keeps track of such types) on return.*

2. *An instance method* `Invoke` *with the signature S. On* call `D.Invoke($S$)` *or* callvirt `D.Invoke($S$)` *an operation equivalent to* `calli` S *is performed.*

3. `ld*` *and* `st*` *as defined in Partition III.*

Continuing our example from above, the OEM would provide for the following description of the vector table:

```
[StaticDelegate]
public delegate void InterruptHandler ();

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

  /* ... */
}

public class H8_3297
{
  [MemoryAlias(0x0000)]
  public static VectorTable VectorTable;

  /* ... */
}
```

The system programmer would then define and register their interrupt handlers like that (in C# 2.0):

---

[3]Delegates are function pointers in ECMA 335. They can point to static or instance methods.

```
[InterruptHandler]
static void Handler ()
{
  /* handler code as usual */
}

public static void Main ()
{
  H8_3297.VectorTable.NonMaskableInterrupt = Handler;
}
```

In C# 1.0, a delegate must explicitly be created in the Main method:

```
public static void Main ()
{
  H8_3297.VectorTable.NonMaskableInterrupt =
    new InterruptHandler (null, Handler);
}
```

2.3 Concurrency

The most basic thing when implementing concurrency is the ability to switch the CPU context. We propose the following mechanism: There is a class `Cpu` that contains a platform-specific value type `Cpu.Context` with no public members. Further, there is a static field `Scheduler` of type `Cpu.Context SchedulerDelegate (Cpu.Context)` in `Cpu` along with two methods `void InvokeScheduler ()` and `Cpu.Context InitContext (ProgramCounterDelegate entryPoint, native unsigned int stack)`—as you might guess, `ProgramCounterDelegate` is a static delegate type. Whenever `Cpu.InvokeScheduler` is called, it completely saves the current context and then calls the method in `Cpu.Scheduler`. As you can see, this method gets a context and returns a context; however, it is not important for this method what a context is. After the method has returned, `Cpu.InvokeScheduler` sets the new context and returns. Due to its signature, `Cpu.InvokeScheduler` can and is intended to be registered directly as an interrupt handler. Since the system programmer shall not know what `Cpu.Context` is, the method `Cpu.InitContext` generates a new `Cpu.Context` instance out of the given `entryPoint` and `stack` pointer. With this mechanism, it becomes possible to implement schedulers in C# independently from the underlying platform.

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

public static Cpu.Context DetermineNext
  (Cpu.Context context)
{
  Cpu.Context newContext;

  /* determine the new context */

  return newContext;
}
```

Furthermore, in any embedded system, programmers need to enter power saving modes. Usually, this highly depends on the underlying platform, but for the implementation of idle threads, we suggest the usage of `void Cpu.Sleep ()`. Moreover, the implementation of synchronization requires to dis-/enable all interrupts at once. For that purpose, we provide for a class `Cpu.Interrupts` with two static methods `void EnableAll ()` and `void DisableAll ()`. Manufacturers can further decide to integrate methods for enabling/disabling single interrupts in this class.

3 Implementation

We built a front end for the GNU Compiler Collection (GCC) [5], [11], thus constructing an ahead-of-time-compiler. This compiler reads in .NET/ECMA 335 binaries and produces target-specific binaries. Our front end's job is to parse the .NET binaries and translate them into an internal tree representation that the GCC's middle and back ends optimize and use for emitting code. In principal, our front end parses only those parts of a .NET executable and the libraries that executable depends on which are directly or indirectly referenced by the executable's entry-point. Hence, the size of generated executables is reduced.

Parsing of methods is done by the virtual execution of intermediate instructions and the construction of respective GCC internal tree along the way. Using the existing back-ends of the GCC, which exist for many processor architectures, we are able to produce highly optimized native target executables.

Alias attributes were straight forward to implement. Whenever a field is attributed accordingly, we do not create the respective variable (for static fields) or field slot (for instance fields) in the GCC front end. Accesses to such fields are realized as the rules might suggest.

The attribute `InterruptHandler` could be directly mapped to the GCC attributes `"interrupt"` and `"interrupt_handler"`. Unfortunately, the GCC's back ends are not consistent in the use of these attributes; some use `"interrupt"`, others `"interrupt_handler"`, and there are also some back ends that do neither. For that reason, we simply attach both attributes to such functions, hoping that respective back ends will be supplemented by either attribute in the future.

If a type is attributed `StaticDelegate` we use simple pointer types. The construct was defined in a way that allows to omit type checks at runtime, so we can use function pointers in lieu of more complicated structures.

Naturally, the implementation of `Cpu.InvokeScheduler` and `Cpu.InitContext` depends on the target. In figure 2, we show an implementation of the former for the LEGO RCX [7] that contains a RENESAS H8/3297 board [9][8]. As you can see, we save the context on the stack, such that we only need to pass the stack

pointer as the "context" to the scheduler method. On return, the new context's stack is then used. Therefore, we need to construct such a stack specifically to the target as shown in figure 3.
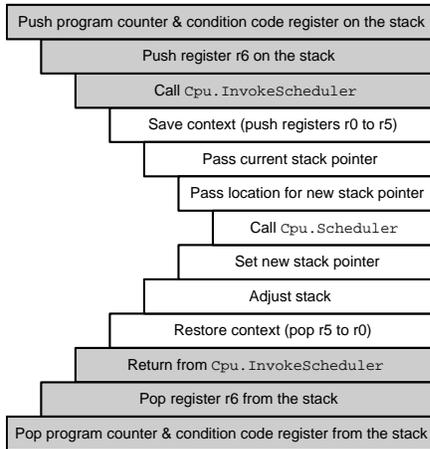

Figure 2. Scheduling on LEGO RCX

Figure 3. Stack during context switch

4 Performance

In a prototypical example, we implemented a sound generating algorithm for the LEGO RCX in C# shown below:

```csharp
public static void Sound (ushort frequency)
{
  H8_3297.Board.Timer8Bit.Channel0.ControlRegister = 0
      x00;

  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 = 0
        x0B;
  }
```

```csharp
  else if (frequency <= 488)
  {
    /* ... */
  }
  /* ... */
  else
  {
    H8_3297.Board.Timer8Bit.Channel0.ConstantRegisterA =
        (byte) (4000000 / frequency);

    H8_3297.Board.SerialTimerControlRegister |= 0x01;
    H8_3297.Board.Timer8Bit.Channel0.ControlRegister = 0
        x09;
  }
}
```

Altogether, the code just presented has seven paths. For every path, we calculated the overhead our code contains (according to [8, Appendix C]). We found out that - in an ideal solution - some of the generated instructions are unnecessary. Table 1 shows our results; if we consider the paths uniformly distributed, we can average the execution overhead for that function to around 24.0 %. While we are not quite sure whether you can really say that our overhead is one quarter, we think this gives an impression that the generated code is not too bad, notabene in comparison to an *ideal* solution.

Path	Cycles (our)	Cycles (ideal)	Overhead
1	137	89	53.9 %
2	247	193	28.0 %
3	277	223	24.2 %
4	343	289	18.7 %
5	371	319	16.3 %
6	401	343	16.9 %
7	389	354	9.9 %

Table 1. Comparison of CPU cycles in every path of our program to an ideal solution

Further, the InterruptHandler attribute is translated optimally by the GCC (at least for the H8/300), ie only those registers are saved on the stack that are really used by the handler method. Moreover, static delegates are translated to simple pointers.

With respect to the speed of our compiler, we found out that the whole approach - writing code in C#, translating it to the CLI, and then to a target platform - seems quite promising. When we ran the sound generating example above, the whole process (from the C# source to the completely linked H8/300 binary) turned out to be faster than compiling and linking the functional equivalent in C.

We conducted both tests 4000 times, which can be considered a test base large enough for confident results, on a machine containing an AMD ATHLON 2000 and 512 MB RAM, using CYGWIN on MICROSOFT WINDOWS XP. For the C# compiler we used the one included in MICROSOFT's .NET framework version 2.0.50727; assembler and linker stem from the GNU BINUTILS collection version 2.16.1. Times were taken by the BASH built-in command time.

With respect to the dimension of our measurements, this is sufficiently precise. For evaluating the results, MICROSOFT EXCEL 2003 was used. The experimental average execution times were 4.63 for CLI/C$^\#$ and 6.48 for C with experimental standard deviations of 0.064 and 0.099, resp. The minimal and maximal values were 4.53 and 4.98 compared to 6.21 and 6.94. A graphical display of the frequency distribution of execution times for all runs can be found in figure 4.
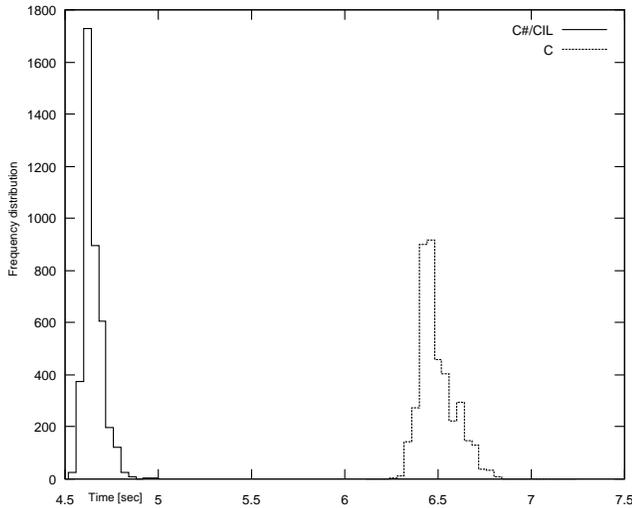

Figure 4. Execution times of compiler runs

5 Alternative Approaches

We deliberately did not follow the approach of allowing access to addresses that are determined at runtime such as represented by the methods `ReadByte` and `WriteByte` in the class `System.Runtime.InteropServices.Marshal` of MICROSOFT's implementation of the CLI in the .NET FRAMEWORK, the classes `IoPort` and `IoMemory` of the SINGULARITY project [10], or the REAL-TIME JAVA class `javax.realtime.RawMemoryAccess` [4]. This is merely pointers in disguise that could result in wild pointers by-passing security mechanisms and compromising analyzability of programs. Moreover, we do not see any necessity of accessing particular addresses other than for interaction with the underlying platform.

For the same reasons, we revoke the proposal of using `unsafe` sections in C$^\#$ code [6][2, p 417]. In particular, the C$^\#$ standard explicitly states that the possibility of introducing *unsafe* code is to be considered a *safe* feature because the unsafe code is clearly marked so then. This is a rather weak explanation, and therefore we prefer extensions that allow to write safe (in the sense above) code.

Furthermore, our approach has several advantages in comparison to using a proxy. First of all, a proxy has to

be generated. This is semantically imprecise as the hardware is not to be created once our programs runs on it; and besides the memory consumption of a proxy object whose only task is to forward access to other locations, the address of the object had to be specified at *runtime* allowing for arbitrary memory access. Additionally, the implementation of proxies is a much harder and thus more error-prone task than our approach.

There are linkers that allow the placement of certain variables at certain memory locations. Our memory alias attributes could have been modeled using this mechanism; instead of applying an attribute to the field in question, one declares the field as external and defines its address in a linker script. Our approach has the advantage of providing a uniform mechanism for accessing hardware as well as means for suitable structural abstractions. Additionally, linker scripts are linker specific, whereas with our attributes the OEMs have to specify their hardware only once.

In ADA, it is possible do declare addresses of variables in the source code [1]. This is essentially the same as our `MemoryAlias` applied to a static variable. There is no equivalent for port-based I/O. As with linker scripts, our approach is more uniform and allows for better modularization.

We are aware that our approach does not allow for *every* speciality that could arise from interacting with hardware. If, for example, a memory-mapped register can change its address arbitrarily at run-time there is no way to deal with this using our fixed-address attributes.[4] However, the advantages (analyzability, safety and readability/maintainability) overcome these restrictions easily, especially when considering that such specialities are rather rare in embedded systems.

Moreover, we did *not* include an implicit definition: If a type contains fields attributed with `MemoryAlias` then a field of this type, that does not bear a `MemoryAlias`, could have an implicit `MemoryAlias(0x00)` attribute, thus guaranteeing a base address for calculation of absolute addresses. For consistency reasons, we decided against this.

There were suggestions to restrict the use of both alias attributes to static fields only and to use two attributes `MemoryAliasOffset` and `PortAliasOffset` for instance fields because it was claimed that it might be an advantage for persons who do not want to spend too much time on learning the whole concept. It was argued that it might not be intuitionally clear that the original attributes applied to static fields have a different effect as if applied to instance fields. On the other hand, this would sacrifice the minimalistic approach, introducing another construct where it is not absolutely necessary. Since an alternative implementation is not very hard to realize[5], we decided to leave this until it actually turns out to be necessary.

---

[4]If in contrast there are only a limited number of such memory locations, we could specify all of them and then use the appropriate one.

[5]Essentially, the only thing you have to do is to check for wrong attribute usage which is quite easy a task to perform.

The only alternative we can see to our approach to interrupt handling is the usage of custom, platform-dependent runtime functions.

Probably, people would like to hide the context switch behind a platform dependent, static property `Hardware.Cpu.CurrentContext`. Context switches must then include a call to the getter method to save the current context and a call to the setter method to set the next context. Then we could write code like this:

```
using Hardware;
using Renesas;

class Scheduler
{
  public static Context currentContext;

  public static void TimerHandler ()
  {
    Cpu.CurrentContext = Scheduler.determineNextThread (
        Cpu.CurrentContext);
  }
}
```

We decided against this because this would imply a freedom that the programmer does not have. The context must always be read in at the beginning of a method and can only be set at its end.

6   Summary & Outlook

We presented a new extensions to the standard ECMA 335. First of all, we showed a general approach for mapping hardware registers not only to simple variables but to structured value types for both, memory-mapped and port-based I/O. The introduction of appropriate attributes and an enhanced redefinition of value types semantics and related instructions, enables OEMs to specify their hardware in high-level languages such as $C^{\#}$.

Further, we seized and formalized the idea of attaching an attribute to a method in order to make the compiler generate code suitable for immediate interrupt handlers. By reducing the generality of delegates, our concept of "static delegates" allows for the integration of external locations for function pointers with the CLI. With both mechanisms, it is possible to write the whole interrupt handling in high-level languages.

Moreover, we were putting things forward towards the implementation of an operating system by introducing a general concept for context switches. Separating scheduler, timer interrupt handler, and context switch, this concept is a flexible means for implementing many different kinds of concurrency.

By generating considerably efficient assembly code, our proof-of-concept, ahead-of-time compiler demonstrates, that programs for virtual machines can appropriately be written even for hardware-near programming.

The opportunity to separate the specification of hardware locations from their use in applications allows more transparency for users (ie system designers) of that code; compilers could be enhanced to disallow certain parts of a program to use memory alias attributes on fields, thus making it possible to restrict the specification of hardware locations to trusted OEM libraries. Also, analysis tools can effortlessly find out, which hardware locations are accessed, eventually enabling programmers do find bugs faster and giving users more insight about the programs they use. Even the code itself becomes more readable and hence better maintainable because programmers are forced to use descriptive names in lieu of cryptic numbers then.

With respect to our front end and the GCC in general, we suggest various optimizations. For example, the concept of interrupt handlers should be generalized such that all back ends implement the same attribute (possibly allowing both attributes for *all* back ends). Additionally, it would be nice to have either special tree nodes or built-in functions for port based I/O operations and the sleep instruction(s), to avoid overhead by linking the generated object files to special runtime files. Another—much harder—task would be an optimized integration of context switches and context initialization.

As another extension, we deem it necessary to provide for attributes that allow programmers to specify certain segments that variables and code should be placed in. Sensible usage includes the definition of ROM code and code that shall run in memory areas that are faster accessible than others. Finally, there should be some mechanism that allows the implementation of memory management modules.

References

[1] Ada reference manual, http://www.ada-auth.org/arm-files/rm.pdf, 2001, 2001.

[2] c# language specification, jun 2005. ISO/IEC approved 2nd edition as ISO/IEC 23270.

[3] Common language infrastructure (cli), jun 2005. ISO/IEC approved 2nd edition as ISO/IEC 23271.

[4] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, and R. Belliardi. *The Real-Time Specification for Java*. The Real-Time for Java Expert Group, 2000.

[5] Gcc home page, http://gcc.gnu.org, 2006.

[6] M. H. Lutz and P. A. Laplante. Ieee software: Real-time systems - c# and the .net framework: Ready for real time? *IEEE Distributed Systems Online*, 4(2):74–80, 2003.

[7] K. Proudfoot. Rcx internals, 1999.

[8] Renesas. *H8/300 Programming Manual*, 1 edition, Dec. 1989.

[9] Renesas. *Renesas Single-Chip Microcomputer H8/3297 Series Hardware Manual*, 3 edition, Apr. 2003.

[10] Microsoft research singularity project, http://research.microsoft.com/os/singularity/, 2006.

[11] R. M. Stallman and et. al. Gcc internals. Technical report, Free Software Foundation, 2005.

[12] M. von Löwis and A. Rasche. Towards a real-time implementation of the ecma common language infrastructure. In *ISORC*, pages 125–132. IEEE Computer Society, 2006.

[13] W. Wolf. *Computers As Components: Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers, 2001.