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

Martin v. Löwis and Andreas Rasche
Hasso-Plattner-Institute at University of Potsdam, Germany
{martin.vonloewis|andreas.rasche}@hpi.uni-potsdam.de

Abstract

In the development of embedded systems, higher-level programming languages have become popular because they often induce higher developer productivity. Tool developers desiring to offer support for such languages need to solve various problems; in particular, they need to support the creation of real-time systems. In this paper, we present an approach for supporting real-time capabilities in C# and, consequently, the Common Language Infrastructure.

1. Introduction

In embedded systems development, various aspects have to be considered in addition to the issues encountered for desktop applications [21]:

- The systems often have limited support for interaction with the developer and for debugging.
- The systems often interact with "special" hardware, which require extra consideration in the control algorithms; controlling this hardware often is the primary purpose of the embedded system.
- The systems often have limited hardware resources: not only is the memory limited and the CPU frequency low, but other constraints, such as energy consumption and space, impact the system development as well.
- The systems often have real-time requirements, meaning that they have to issue control commands to the hardware before a certain deadline.

Many approaches have been taken in the past to address these issues and to simplify the development process for such systems. Still, software technology is always behind those on desktop system, as the more advanced technologies require more hardware resources or otherwise are unsuitable for embedded systems.

As a specific example, the ECMA Common Language Infrastructure (CLI) [5], as implemented in the Microsoft .NET framework [15], is typically implemented by means of just-in-time compilation, and using garbage collection. Neither of these techniques is suitable for real-time programming. Just-in-time compilation causes hard-to-predict execution times for a method, as execution time varies depending on whether the just-in-time machine code is available or needs to be generated. Likewise, memory allocation actions, such as object creation, vary depending on whether garbage collection is invoked to free unused memory for reuse.

On the other hand, the CLI would offer the developer of the embedded systems developer the same advantages that they get on desktop systems:

- due to the predictable nature of the sandbox-mode execution of CLI instructions, programming errors never result in system crashes, but cause exceptions to be thrown. This allows for a simpler post-mortem analysis of a fault.
- due to the support for rapid prototyping, simulators for the target can be more easily created. Ideally, much of the code would only use standard library functions of the CLI, so that simulators are only necessary for the target-specific hardware.

We are currently working on an implementation of the CLI for embedded systems, initially targeting the Lego™ Mindstorms™ hardware. To support the CLI on such a system, we need to solve the following aspects:

- 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.

In this paper, we give an overview of the first two aspects, and then focus on the real-time properties.

2. Intermediate Language Execution on Embedded Processors

The first challenge for executing CLI programs is the Intermediate Language (IL) code that is used to represent CLI algorithms. This intermediate language was designed to be independent of a specific execution platform, with the expectation that just-in-time (JIT) compilers are created for all platforms on which CLI code is to be executed.

Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing
0-7695-2561-X/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

Creating a JIT compiler for a processor is no small task, though, and pays off only if a large number of different programs are written for the processor. While this is the case for the Intel x86 processors for which CLI implementations are readily available, no such implementations are available for the majority of microprocessors used in embedded systems.

In addition, even if a JIT compiler is available, it is quite expensive to use: Running it on the target machine takes a lot of processor cycles, and makes prediction of execution times difficult. Even if the compiler would produce target code at application start-up time, code quality of the JIT compiler often is worse than the quality of an optimizing compiler.

We have taken a different approach for the execution of IL code, as shown in figure 1.

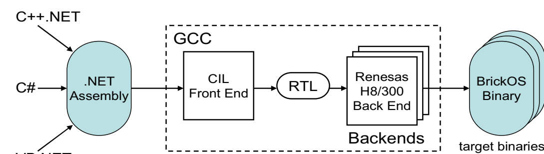

Figure 1 : Binary Translation of IL-Code with the GCC

We have modified the GNU Compiler Collection to support compilation of IL code into target machine code. This approach is similar to the gcj compiler [8]. We compile each method in intermediate language into one function on assembler level, using symbolic execution to translate the stack machine instructions into GCC's internal statement representation, which is then optimized and compiled into assembler code.

2.1 Integrating with the Target

This compilation process is only one part of the entire implementation of the CLI. In addition, we need

- an implementation of a run-time support library for functions used specific IL opcodes,
- an implementation of the standard CLI library, and
- an implementation of target-specific interfaces.

The run-time support library primarily deals with memory management and exception handling. For example, the IL code supports a newobj opcode, which creates an instance of a given class and invokes the constructor. The GCC will compile this opcode into a call to the newobj run-time function, followed by a call to the constructor (passing the address of the newly-created object as first, implicit parameter). The implementation of the newobj function is target-specific; it needs to allocate memory of the size of the object, and zero-initialize this memory. If the system runs out of memory, the function needs to throw an exception.

The standard CLI library provides the classes defined in the ECMA standard, and available in most CLI implementation. This library is quite large, providing a lot of functionality that likely isn't needed in the embedded application (like reflection, or code access security). We take the approach of only including those parts of the standard library which are actually needed by the application, reducing the size of the resulting executable as much as possible. This is done through standard linker semantics: if the application references some functionality of the library, the linker will drag-in the corresponding implementations. Work is in progress to improve this selective inclusion: for example, inclusion of a class currently means that all virtual functions of that class are included, when it might be sufficient to omit some of them (e.g. when it is possible to prove that they are not called).

In many cases, functions in the standard library can be implemented in IL code; this is an approach that most CLI implementations take. We follow that approach wherever possible, and then compile these implementations to native code just as we do with the rest of the application.

In addition to the standard library, the special hardware must be made available for CLI applications. In many cases, this hardware can be exposed as another library. For example, when targeting the Mindstorms™ RCX, we build our implementation on top of the brickOS operating system [3], and expose many of its functions as CLI functions, in a predefined class brickOS (with classes such as dmotor, dsensor, or dsound). In exposing such functionality, there is often the choice of implementing a standard library function in terms of target-specific hardware, or wrapping the target functionality in a separate API. In these cases, we favour usage of the standard library API, as this will likely improve the portability of libraries and smoothen the learning process for new developers.

Certain aspects of target integration are still subject of ongoing research. In particular, some functionality cannot be readily translated into a class interface, e.g. reaction to hardware interrupts. While traditional solutions (such has polling or blocking APIs) have worked so far, a more direct solution (e.g. being able to implement an interrupt handler in C#) would be desirable.

3. Real-Time Properties

To support real-time programming, several techniques have emerged. We consider the following aspects most relevant, and have integrated them into our implementation:

- predictability of execution times; in particular, of the worst-case execution time.
- multi-threading and scheduling; in particular, control over thread-priorities

While we are aware that different approaches at constructing real-time systems exist, we believe that

IEEE COMPUTER SOCIETY

these two aspects together already allow the construction of meaningful real-time application. Consequently, we will study them in the following sections.

3.1. Worst-Case Execution Times

To predict the execution time of a program (at least in the worst case), the code of the program must be available. As we are implementing only the programming language, we clearly cannot give predictions for arbitrary programs. However, if the program code is available, prediction of execution times is simplified if the execution times of certain basic operations are known.

In many practical applications, a precise prediction of the worst-case execution time (WCET) isn't necessary; instead, knowing an upper boundary is sufficient to guarantee that the deadlines are met.

In our implementation of the CLI, we can guarantee that certain byte code operations will always execute as a constant sequence of machine instructions. Traditional techniques for inferring worst-case execution times apply [20, 7]: If a performance model for the processor is available, the actual machine code could be analysed. Otherwise, test runs can be performed for individual instructions, and upper boundaries for the worst-case execution time can be inferred.

Constant-time execution can only be guaranteed for certain operations; for other operations, execution time depends on application parameters. In the subsequent discussions, we discuss the guarantees that we can provide for the various operations.

As a basis for the following discussions, we first need to present the principles for representing data (classes, instances, …) at the run-time of a CLI program:

- integral types are represented using the machine representations; the CLI specifies the widths of the various types (e.g. C# int denotes a 32-bit type independent of the target machine)
- floating point types are similarly represented, typically in IEEE-754. If the target hardware does not support floating point directly, an emulation library can be used. In this case, usage of floating-point operations is best avoided in the application

Currently, not all opcodes of the intermediate language have been implemented; the reader should assume that opcodes not mentioned in the following sections are unimplemented. The set of implemented opcodes is primarily oriented towards those emitted by the Microsoft C# compiler.

3.2 Operations Executing in Constant Time

As a rule, all operations that are "directly" supported in hardware execute in constant time. More specifically, these are:

- loading immediate arithmetic values (ldc.i4.*, ldc.i8.*)
- access to local variables (ldloc.*, ldarg.*, ...) except for C# structs (for which the load operation may depend on the size of the struct)
- access to class and array members (ldelem.*, ldfld, …)
- store operations for primitive data (numbers)
- branch instructions (br.s, brfalse.s, beq.s, bgt.s, ...) except for switch and exception handling,
- method calls (call, callvirt, …)
- arithmetic operations, including conversions (add, sub, mul, conv.i1, …)
- object type casts (castclass, isinst)
- string load operations (ldstr)

While this list may sound plausible for most of the operations, several entries deserve further explanation.

It might be surprising that store operations have constant time only for primitive data. Clearly, user-defined value types (C# structs) take time depending on the number of bytes to be copied. However, storing an object reference may also consume varying time: as discussed later, assigning to a variable may trigger a memory deallocation of the prior value of the object, which may cause further operations to be performed.

Arithmetic operations typically take a constant time even if they are not directly supported in the hardware. For example, on a 16-bit processor, 32-bit addition is not directly supported. However, the compiler will generate a fixed sequence of machine instructions to implement the addition (as two 16-bit additions), resulting in a constant execution time for the entire operation. This even holds if the implementation is performed in a library routine. For example, in an IEEE-754 emulation library, operations can be implemented in a way that a predictable worst-case execution time can be guaranteed.

For a method call in a programming language, the execution time typically depends on the number of parameters, as each parameter needs to be computed. However, in IL code, the parameter computation can be accounted for in the opcodes that push the parameters onto the evaluation stack; the call instruction then becomes a simple, single-instruction control transfer. Further consideration is necessary for virtual methods: the target address of the control transfer depends on the type the object. In this case, we use the concept of virtual method tables [4], where we statically compile a method number into the machine executable. While this allows for constant execution time of a call to a virtual method, we limit support for separate compilation: whenever a class changes, all

CLI classes that inherit or use that class need to be recompiled. For an embedded target, this is typically not an issue, since the entire application code can be assumed to be available at deployment time.

For interface calls, we plan to use the same approach to constant-time method dispatch as the Mono project [14].

The reader should observe that we only considered the time for the actual control transfer here: the total time for the method clearly depends on the code inside the method. In the presence of virtual function, this may be difficult to predict, as it is not statically clear which method is called. As a first approximation, the maximum of all implementation of the virtual functions could be taken; in specific cases, it might be possible to exclude certain virtual function implementations for certain call sites.

For class casts and type membership tests, a naïve implementation would normally require a time linear to the inheritance depth. Again, borrowing ideas from the Mono project [14], constant time casts (both to classes and to interfaces) becomes possible.

String load operations can be implemented in constant time, because the compiler already arranges the bytes of the internal string representation in a way that requires no copying at run-time.

It is worth pointing out that the constant-time guarantees only hold for the non-exceptional case. Exception handling is discussed later.

3.3. Memory Management Operations

A number of operations affect memory management. In our CLI implementation, these are

- newobj (creation of a new instance)
- newarr (creation of a new array)
- box (creation of a value box)
- assignment operations to object types (stloc.*, stelem.*, starg.*, stind.*, stfld, …)

The worst-case execution time depends on two factors: the time for the memory management itself (allocation, deallocation), and the time for initialization/finalization.

Of these, initialization time is easiest to predict: as each object must be zero-initialized in the CLI, initialization takes time **linear** to the number of bytes in the object. This is a per-class constant; for arrays, it depends on the number of elements in the array. For classes, the low-level initialization is followed by a constructor invocation. This should be accounted-for like a method call: the transfer to the constructor takes a constant time; the total time for the constructor clearly depends on the code in the constructor (compared to method calls, virtuality does not complicate the analysis here, as constructor calls are always non-virtual).

For allocation of memory, various allocators have been discussed in the past [13,22]. In the case of real-time CLI implementations, we consider memory pools as the most predictable approach. In our implementation, we currently use the following approach:

- Object sizes are grouped in size classes, typically spaced 8 bytes apart. There is a maximum block size beyond which no pools are maintained; this is a tunable parameter.
- Per size class, there is a singly-linked list of free memory blocks of that size class.
- In addition, there is a global pointer to free space.
- On allocation, the requested size is rounded up to the next size class.
- If the list of blocks of the size class is not empty, the first one is removed. If it is empty, a block from free space is taken. This is done atomically (using thread locking), to avoid race conditions if multiple threads allocate simultaneously.
- On deallocation, the block is returned into the front of its size class, again using thread synchronisation.
- To avoid priority inversions if a low-priority thread is interrupted while holding lock, the lock must be associated with a priority boost [9,18]

This implementation provides for **constant-time memory allocation and deallocation**. As memory is always returned into the pool of its size class, memory fragmentation can occur where the memory is exhausted for some size class, even though sufficient space would be available in a different size class.

It is also worth noting that the timing guarantee only holds for blocks below the maximum block size: larger blocks are managed with a different allocator, for which no timing guarantees can be made. This typically isn't a problem, since such larger blocks are of unpredictable size, anyway, and initializing them will consume an unpredictable amount of time.

As the CLI does not support an explicit memory release operation, releasing memory must be done automatically. Most CLI implementations rely on garbage collection algorithms, which have been shown not to be suitable for real-time requirements [10]. In our implementation, we rely on **reference counting** [10] instead: each object maintains a reference counter, which is incremented whenever the reference is stored in a variable, and decremented whenever that variable is assigned a new value. Care must be taken in the presence of multi-threading: we rely on the hardware's support for atomic increment and decrement operations; if these are not available, explicit thread locking again becomes necessary. Again, these operations take **constant time**.

When the reference count drops to zero, the object is no longer referenced, and its memory can be deallocated; as discussed above, this, in itself, takes constant time. However, the object released might refer

IEEE COMPUTER SOCIETY

to other objects. Dropping the references to these objects requires us to decrement their respective reference counters. This, in turn, could result in further objects becoming unreferenced. Therefore, the time for **object deallocation is linear** with the number of objects that become unreferenced. As each assignment could, in principle, cause some object to become unreferenced, all assignments (to object variables) can cause such linearly-timed computation.

Still, reference counting remains highly predictable: from the application logic, it will be typically clear when an object becomes unreferenced, and how many objects become unreferenced subsequently. If the deallocation overhead is unacceptable at some point in the code, the old objects could be queued in a separate data structure before being replaced, so that they don't become unreferenced when they become unused. A low-priority thread could then clear the queue, causing the actual deallocations to occur.

A well-known limitation of reference counting arises from **cyclic data structures**: an object cycle could become unreachable, yet its reference counts would not drop to zero, and the objects would become garbage. This is a deviation of traditional CLI implementation; however, the problem can be solved in the application: if the application developer is aware of that limitation, she can explicitly break the cycles (by assigning null appropriately) before dropping the last reference to the cycle.

Yet another challenge arises from **finalization**: In the CLI, each class may implement a Finalize method, which is invoked when the object's memory is reclaimed. We currently envision two strategies to deal with finalization: First, we could invoke the finalization immediately when the last reference to the object is dropped. That would make predictions harder, as any object assignment may not only trigger deallocation, but also finalization. The other approach is to queue finalizable objects in a separate queue (which can be done in constant time), and perform finalizations in a separate thread; this is the approach that most other CLI implementations take. We expect that different applications have different needs here, and plan to offer a choice.

3.4 Exception Handling

The CLI provides an elaborate notion of exception handling, where both the CLI implementation and the application can throw exceptions at any point in time, causing a change in control flow in all callers until the exception is caught, or the program is terminated. In particular, the following operations can cause exceptions to be thrown in the CLI implementation:

- field or array access, if the object reference is null
- array access, if the index is negative or past the end of the array
- array stores, if the type of the object being stored does not match the type of the array
- virtual function calls (callvirt) if the object reference is null
- arithmetic operations, for overflow, division by zero, etc.
- object allocation, if the system runs out of memory
- cast exceptions, if the object cannot be cast to the target type

This list is incomplete; for a complete list of all cases in which the CLI implementation will throw exceptions, see [5].

In our current implementation, exception handling is not yet implemented; instead, an exception causes the immediate termination of the program. Even if the exception handling mechanism was implemented, real-time guarantees could not be made, as it is very difficult to predict how control will flow when an exception occurs. Most of the conditions listed above are considered programming errors; if it can be shown (e.g. by proof) that they cannot occur for some application, prediction of worst-case execution time becomes unnecessary.

For user-defined exceptions, the situation is similar: in most cases, it is possible to check whether an exception would occur before it actually does; APIs should be designed with that goal in mind.

It should not be concluded that exception handling is completely unnecessary in an embedded environment. To the contrary: as discussed in the introduction, analysis of errors in a system become much easier if precise exception handling is part of the platform. Furthermore, ESA studies [11] have shown that the proper reaction to an error in the embedded system is not the system shutdown, but reconfiguration into a fallback controller, perhaps with a degraded functionality.

3.5 Library Functions

Perhaps the biggest challenge in execution time prediction is the prediction of library execution times. This is particularly true for the CLI, which offers a huge variety of API, for various tasks. In many cases, time complexity depends on the amount of data being processed, starting with operations as simple as string concatenation, up to complex libraries such as XML processing.

For operations occurring in our applications, we have analysed the worst-case execution time, often aiming for an implementation in constant time. In general, we try to implement library functions by means of IL code; this strategy allows for an analysis on the level of the intermediate language, instead of analysing for the specific target.

IEEE COMPUTER SOCIETY

4. Multi-Threading and Scheduling

For a real-time application, tasks are typically performed in separate threads. Priorities are assigned based on a priori knowledge of deadlines and execution times of the tasks, and the scheduling algorithms used. This notion of system design directly transfers to the CLI, except that some details of the integration into the standard library need further consideration.

In our implementation, we assume that a real-time operating system for the target is already available, and that this system provides control over multi-threading and scheduling as needed.

4.1 Threads

The standard CLI library already includes a thread library (*System.Threading*) which can be mapped to system threads. To do so, native code must be integrated into the library implementation which creates a system thread whenever a CLI thread is started; this system thread then will need to run the thread code, represented through a CLI delegate.

One challenge in this implementation is the *CurrentThread* property, which returns a reference to the CLI thread object. As this is a frequent operation, execution in constant time is desirable. This can be achieved if the system supports thread-local storage, and if the thread-local storage in turn provides constant-time access. Most systems do have this property, however, to understand the performance implications of the seemingly-innocent property access, careful analysis of the target operating system is necessary.

4.2 Thread-local storage

On top of the threading API, the CLI offers thread-local storage. In most implementations, such as the Rotor framework [12], this API relies on a hash table. While a hash table offers good performance on average, it makes no guarantees for worst-case execution time (in case of hash collisions). Fortunately, the CLI threading API also allows for an implementation based on indexing, assuming an upper boundary on the number of LocalDataStoreSlot instances can be given. This allows for a trade-off between expressiveness and predictability.

4.3 Thread Priorities

In most real-time systems, a fairly large number of priorities are available; yet *System.Threading* only distinguishes five thread priorities. Developers commonly rely on many more priorities to design schedules; offering a CLI API for more priorities is thus essential.

Fortunately, *System.Threading.ThreadPriority* is an enumeration based on the int type, allowing to define additional enumeration values without breaking the type safety of the C# or intermediate languages. Our current approach is to assume that the five predefined priorities (*Lowest, BelowNormal, Normal, AboveNormal, Highest*) are all below the real-time priorities, which the application developer needs to assign explicitly, using values from a target-specific API.

4.4 Periodic Threads

Typical real-time control applications include cyclic activities e.g. for data sampling. Most real-time operating systems provide a periodic thread API which allows for creating periodic threads. Threads can use a scheduler method *WaitforNextPeriod* to synchronize with the specified period. We will add periodic threads to the class library by providing a *System.Threading.PeriodicThread* class for thread creation and a static method *System.Threading.PeriodicThread.WaitforNextPeriod* the implementation of periodic threads.

4.5 Extending Thread Synchronization

The priority ceiling protocol [18] can be used to avoid the priority inversion problem. In order to use this protocol a ceiling value must be provide when creating a synchronization object. We have extended the constructor of the *System.Threading.Mutex* class allowing the definition of a ceiling value.

5. Performance Evaluation

We have used our implementation for our experiments within the Distributed Control Lab [17]. The "Higher Striker"-Experiment relies on a configurable dual-ported hardware buffer in order to deal with varying jitter in used control applications and operating systems. The experiment hardware periodically reads one entry from the buffer with a frequency of 38400 kHz. The control application can use a configurable buffer size ranging from 0 to 255 entries. Depending on the buffer size used, the application has more or less stringent deadlines for buffer refill: if we start off with the buffer filled, buffer refill can be delayed longer. An empty-flag of the hardware buffer indicates an error during execution. Using this experiment hardware we are able to measure real-time performance of used control applications by evaluating the buffer size, that must be used during program execution. The experiment gives an indication of the average jitter of the control application.

We have evaluated our compiler front-end based on the GCC 3.4.1. We have implemented a runtime

IEEE COMPUTER SOCIETY

library for the Windows CE.NET 4.2 operating system, which provides support for threads, synchronization and memory management. Experiments have been performed on an Intel Celeron 633 MHz with 256 MB RAM.

We have evaluated 3 versions of the control application. One has been implemented and compiled using the .NET Compact Framework. A second program has been also implemented using the .NET Compact Framework, but was manually manipulated to set the thread priority to real-time (highest available). Lastly we used our approach for the execution of the .NET program.

Within the experiment we increased used buffer size from 0 to 100 and checked for errors (empty buffer) in the experiment hardware. Figure 2 shows our measurements for 100 runs of the experiment described.

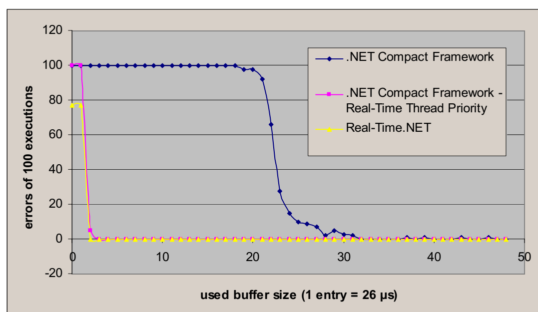

Figure 2: Average jitter during execution

The pure .NET Compact Framework application performs worst. Using 18 buffer entries or less, all runs of the application fail. When using 30 entries and more almost all runs complete successfully, except some few runs that fail also when running at a higher buffer usage.

Our implementation Real-Time.NET produces no error when using at least 3 buffer entries. There are also some successful runs (23) when using less than 3 buffer entries.

Lastly the .NET Compact Framework application with manipulated thread priorities performs good as well. This demonstrates that our approach of extending the .NET library API to support real-time aspects is both necessary and sufficient in this context: In the specific application, the primary reason for not meeting the hard deadline is that some other system activity is in our way. By increasing the priority of our task above those of the system tasks, we can meet all deadlines.

One may wonder what the effect of predictability is, in this case: if the modified Compact Framework also solves the task, isn't that just as suitable for the real time problem? We argue that it is not: while the Compact Framework comes out responsive in our test runs, we have no way of knowing whether it is predictable in its behavior. We have demonstrated how worst-case execution times can be established in our approach, providing higher trust that the experimental results will indeed be repeatable.

The measurement do not show any influence of the garbage collector and the just-in-time compiler include in the .NET Compact Framework. The just-in-time compiler is only activated at the first run of a method, but the measurements have all been taken during one experiment run (the process has been started once). There was also no garbage collector activity because in the .NET Compact Framework it is activated only if the memory size reaches 750 kBytes. In case of these influences the jitter of the .NET Compact Framework is much higher.

6. Related Work

The Real-Time Specification for Java [1] defined by the Real-Time Java Expert Group specifies extensions to the Java Virtual Machine and additional classes for the Java Class Library in order to allow for a predictable execution of Java applications. These extensions include periodic threads, user-definable scheduling algorithms, asynchronous event handling, raw memory access, priority-inversion-free synchronization and additional concepts for memory management. In order to deal with unpredictable interruption through a garbage collection, scoped memory is introduced which can be used by hard real-time threads and is not subject to garbage collection. These scopes are lexically marked by programmers; memory is released when the scope is left.

There are several implementations of the RTSJ available including the reference implementation JTime[19] by Timesys which interprets Java Bytecode. Timesys implemented all mandatory features of the specification. In order to use all features Timesys' commercial real-time linux implementation must be used.

Sun Microsystems is working on an implementation the RTSJ as well. The Mackinac project extends the Hotspot Virtual Machine Implementation and uses Just-in-Time compilation. There is no implementation downloadable yet, but details of the implementation have been published in [2]. The target operating system is Solaris 10.

jRate is an extension of the GNU gcj compiler frontend and runtime which adds support for most features defined by the RTSJ. Java source files including used real-time classes are compiled into native target code using the jRate compiler. This approach is very similar to ours; instead of the CLI, it targets the Java language and and focusses on POSIX operating systems.

The JamaicaVM developed by aicas GmbH includes a builder tool for integrating Java Bytecode and an corresponding Virtual Machine implementation into a single executable application binary. ByteCode is embedded as C-Array definition and linked with the JamaicaVM library. The implementation supports the

IEEE COMPUTER SOCIETY

extensions defined by the RTSJ including a real-time garbage collector.

Microsoft's .NET Compact Framework is a limited version of the .NET platform for mobile and embedded devices. While there are no time guarantees for any Compact Framework threads, real-time specific code can be implemented within native libraries and invoked by the native call (platform invoke , *PInvoke*) feature supported by the Compact Framework. Using this method at least non-real-time critical sections can be written using a .NET language.

Acknowledgements

We would like to thank Jan Möller for his contributions to the CLI GCC front-end.

7. Conclusions and Future Work

We have presented an approach for real-time applications based on the ECMA Common Language Infrastructure. In our prototypical implementation, we are able to target different CPUs and operating systems, and we are available to provide application developers with execution time predictions for the basic operations, in terms of the CLI intermediate language.

Further work is necessary before this technology becomes suitable for larger applications: our prototype implementation must be extended, to offer more of the CLI functionality (in particular in the area of the standard library). We would like to experiment with various aspects of the programming API, in particular:

- support for driver development (including interrupt handling and hardware I/O) in C#
- support for dynamic reconfiguration of systems, based on the Adapt.NET [16] framework
- integration of whole-program analysis tools to reduce the set of library functions required to run a certain application

In addition, we would encourage authors of tools for worst-case execution time analysis to attempt an analysis on the level of the intermediate language, using assumptions on the execution time of the individual IL opcodes as presented in this paper. We found that the analysis of intermediate language code allows for a high degree of abstraction, and believe that it could potentially allow for target-independent WCET analysis.

Literature

[1] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, Mark Turnbull "The real-time specification for Java" 1st edition, Addison Wesley Longman, ISBN: 0201703238, January 15, 2000

[2] Greg Bollella, Bertrand Delsart, Romain Guider, Christophe Lizzi, Frederic Parain "Mackinac: Making HotSpot Real-Time" in Proceedings of International Symposium on Object-oriented Real-time Distributed Computing (ISORC), Seattle,Washington, USA, May 2005

[3] brickOS Homepage at Sourceforge, "Open source operating system for Lego™ Mindstorms™", http://brickos.sourceforge.net/, 2005

[4] Timothy A. Budd „An Introduction to Object-Oriented Programming", Addison-Wesley 2nd edition, ISBN: 0201824191, August, 1996

[5] ISO/IEC "Information technology — Common Language Infrastructure", ISO/IEC 23271:2003

[6] Angelo Corsaro and Douglas C. Schmidt "The Design and Performance of the jRate Real-Time Java Implementation" On the Move to Meaningful Internet Systems, Confederated International Conferences DOA, CoopIS and ODBASE 2002, ISBN: 3-540-00106-9, Springer-Verlag, London, UK, pages 900-921

[7] Christian Ferdinand. "Worst Case Execution Time Prediction by Static Program Analysis", 18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 2, 2004, p. 125a

[8] Free Software Foundation "The GNU compiler for the Java Programming Language" http://www.gnu.org/software/gcc/java/compile.html, 2005

[9] John B. Goodenough and Lui Sha. "The priority ceiling protocol: A method for minimizing the blocking of high priority ada tasks" *Ada Letters*, VIII(7):20-31, 1988

[10] Richard Jones and Rafael Lins "Garbage Collection : Algorithms for Automatic Dynamic Memory Management", John Wiley & Sons, ISBN: 0471941484, August 22, 1996

[11] J. L. Lions, "ARIANE 5. Flight 501 Failure", http://homepages.inf.ed.ac.uk/perdita/Book/ariane5rep.html

[12] Microsoft Corporation, "Shared Source Common Language Infrastructure", http://msdn.microsoft.com/library/default.asp?url=/library/en-us/Dndotnet/html/ mssharsourcecli.asp, 2005

[13] Minsky "A Lisp garbage collector algorithm using serial secondary storage" Technical Report Memo 58, Project MAC, MIT, 1963

[14] The Mono Project Homepage, http://www.mono-project.com, 2005

[15] Microsoft Corporation ".NET Framework Developer Center", http://msdn.microsoft.com/netframework, 2005

[16] Andreas Rasche, Marco Puhlmann and Andreas Polze. "Heterogeneous Adaptive Component-Based Applications with Adapt.Net" in Proceedings of International Symposium on Object-oriented Real-time Distributed Computing (ISORC), Seattle, Washington, USA, May 2005

[17] Andreas Rasche, Bernhard Rabe, Martin von Löwis, Jan Möller, and Andreas Polze "Real-time robotics and process control experiments in the Distributed Control Lab" in IEE Proceedings – Software, Volume 152, Issue 5, October 2005

[18] L. Sha, R. Rajkumar, and J. P.Lehoczky "Priority inheritance protocols: An approach to real-time system synchronization" IEEE Transactions on Computers, 39(9):1175-1185, 1990

[19] Timesys "Real-Time Specification for Java Reference Implementation (RI)", http://www.timesys.com/java/, 2005

[20] H. Theiling "Extracting safe and precise control flow from binaries", Seventh International Conference on Real-Time Computing Systems and Applications (RTCSA'00), 2000, p. 23

[21] Wayne Wolf „Computers as Components – Principles of Embedded Computing System Design", Morgan Kaufmann Publishers, ISBN: 1-55860-693-9, 2001

[22] McCarthy "Recursive functions of symbolic expressions and their computation by machine", CACM 3:184-195, 1981

0-7695-2561-X/06 $20.00 © 2006 IEEE