Hardware Support for .NET MicroFramework CLR Components

*Sneha.M, Sri Charanya.D, Usha Bhuvaneswari.R*
*Guided by Prof. Dr.Ranjani Parthasarathi*
Department of Computer Science, College of Engineering, Guindy, Anna University

**Abstract**

Microsoft introduced the .NET Framework, which brings the power and reliability of managed code to scale from large servers to embedded devices. However, many devices are still too small/resource constrained to support these platforms. To serve the market for even smaller devices, Microsoft has developed a new platform Microsoft .NET Micro Framework (.NET MF). Field Programmable Gate Arrays (FPGAs), which are reconfigurable, provide a faster execution environment in addition to low initial cost and minimal usage of silicon space**.** Here, we propose to provide hardware support (on FPGA) for the .NET MicroFramework CLR.

## I. Introduction

From sensor networks to smart watches, there are new applications that require low hardware cost and low power consumption, but would benefit from the flexibility, reliability, code reuse, and great tools of .NET. The .NET MicroFramework is designed to work on devices where hardware capabilities are too limited to be practical with the full .NET or .NET Compact Framework running on Windows XP Embedded or Windows CE.

With the proliferation of embedded devices, the system design paradigm has shifted from the conventional design to hardware-software co-*design*. The focus is on building customized hardware and software. The hardware customization has been facilitated by Field Programmable Gate Arrays (FPGAs). FPGAs, which are reconfigurable, provide a faster execution environment in addition to low initial cost and minimal usage of silicon space. FPGA is an array of logic gates that can be hardware-programmed to implement user-specified tasks. Using FPGAs, one can devise special purpose functional units that may be very efficient for some limited task. It is also possible to customize the entire instruction processor using these devices. As FPGAs can be reconfigured dynamically, it is possible to design customized systems for more complex special tasks at speeds that are higher than what can be achieved with general-purpose processors.

The paper is organized as follows. Section II discusses the past trend regarding the design of virtual machines, and about the related work going on. Section III introduces the .NET MicroFramework, the CLR and its components. Section IV details about our scheme of implementation of the CLR, highlighting the design.

## II. Related Work

The field of virtual machines (VM), and language-independent and platform-independent execution environments has always fascinated language designers and implementers for a long time. Such implementations have a lot of advantages over the native compilers strategy, the main objective of adopting such an approach being portability. Portability is achieved by having the high level code translated to an intermediate form, which a system, usually called the Virtual Machine, translates to the native code of the target architecture. The virtual machine is a software implementation that lies between the application and the operating system. Or in other words, the VM can be thought of as a program that can run other programs. One of the ways of realizing such a virtual Machine system, and the widely adopted one, is by modeling the VM as a stack. The reason for adopting a stack-based architecture is that, any virtual machine, which aims to achieve portability across platforms, cannot make assumptions on the underlying architecture. A stack, on the other hand can offer higher level abstractions and abstract actions can be specified on a stack (push, pop, addtop- two etc.). Thus, it becomes desirable to model VMs as abstract stack machines.

The intermediate code form for Java is the Byte code, and the execution environment is called Java Virtual machine. Many advances have been done to increase the speed of execution. One way to achieve faster runtime is to have dedicated java coprocessors or to have specific processors( Java Processors) that can execute Byte codes. There have also been complete FPGA implementations of JVM[5].

.NET is an upcoming technology that is now given attention by embedded systems groups and reconfigurable computing groups. There is a chip that is capable of running a very restricted subset of

the CLR but the scope of the application is pretty narrow. An implementation of the .NET CLR Execution engine has been done on FPGAs[1]. But this implementation of the execution engine supports a set of basic instructions such as arithmetic, logical and comparison instructions. The engine has been modeled as a stack based machine, the stack being maintained in the hardware. The CLR engine runs on the MicroBlaze softcore processor provided by Xilinx specially for FPGAs. We aim at enhancing the execution engine and providing hardware support for the other features of the CLR.

SPOT (Smart Personal Object Technology) is Microsoft's new entrant in embedded device technology, for task specific devices. It is, in a nutshell, a device based on "Ollie" - a 32-bit ARM7 processor core that runs a very, very stripped down Windows OS. Right in the processor is a very scaled-down Common Language Runtime (CLR) known as TinyCLR. What sets SPOT apart is that it leverages the Rapid Application Development (RAD) capabilities Microsoft provides through managed code and Visual Studio 2005. The TinyCLR has an even smaller footprint than what Windows CE offers, and is designed for devices with limited resources and functionality.

## III. The .NET MicroFramework

The .NET MicroFramework supports a subset of the .NET Framework and runs with or without an underlying OS. So the footprint is smaller. Current configurations of the MicroFramework require about 300K of RAM. The MicroFramework provides a subset of full OS features, so it does not require an OS and is referred to as a 'bootable' runtime. Making the runtime 'bootable' means that booting support, interrupt handling, threading and process management, heap management, and other environmental support functions that are typically provided by the operating system have been added to the .NET MicroFramework so that it can run directly on hardware. The OS services that are needed by the application are provided up through the runtime. In creating a bootable runtime, the .NET MF provides a subset of the OS features directly rather than relying on the underlying OS, the subset being selected as services required to run applications on small devices.

**Common Language Runtime:**

The Common Language Environment (CLR) is the run-time environment of the .NET framework. It manages the execution of code and provides services that make the development process easier. The intermediate code form of the .NET system is called Common Intermediate Language (CIL) or the Microsoft Intermediate language (MSIL). The .NET Microframework CLR is a small, optimized "managed code" runtime based on industry-standard ECMA specifications. When native code executes, the operating system really doesn't know anything about the code and therefore can only execute it blindly. Managed code includes a great deal of information about the code so that the CLR can prevent errors.

**The .NET Micro Framework CLR in detail:**

.NET MicroFramework CLR implements all of the major features found in the full CLR. It omits a few features that were deemed inappropriate for this class of device, and adds a few features that are specific to this class of device. The Micro Framework CLR supports major features like Exception handling, time sliced thread management, Garbage collection and defragmentation of memory. In addition to that, there are other features like the type system - Numeric types, Class types, value types, arrays, delegates, events, references, weak references, Serialization (compact version of the original .NET serialization), Non-incremental Mark and Sweep Garbage Collection and Multiple application domains. It also includes the self describing data, as opposed to separate meta data, so as to facilitate garbage collection.

***Execution Engine:***

As with all managed code runtimes, the .NET MicroFramework CLR provides a virtual machine abstraction that is targeted by the compiler tools from the source programming language. In order to execute programs, a special engine is required to load, prepare, and perform the instructions specified in the program (the Intermediate Language [IL]). One of the key benefits of this design is that it allows for code to be developed independently of the actual hardware on which it runs. Since programs are specified in terms of a virtual machine, the execution engine must translate encoded operations into *native* operations (i.e., instructions that can be performed by actual hardware).

There are two basic approaches (and a myriad of variations) to accomplish this: *interpret* the code as it is executing or *compile* it to native code, as it is needed (Just In Time [JIT]). .NET MicroFramework CLR interprets program code.

***Exception Handling:***

One of the keys to creating a safe execution environment is the ability to effectively handle exceptional runtime situations, which might occur as a result of unforeseen operating conditions. Applications are allowed and encouraged to use the exception handling mechanisms found in traditional .NET programs. The execution engine manages the dispatching and clean up associated with exceptions. In addition, it has special mechanisms for protecting the integrity of the system as a whole from unruly programs, by gracefully cleaning up state that could be causing a program to misbehave. Ultimately, a program will be "blacklisted" if it continues to fail according to a pre-determined heuristic.

***Garbage Collection:***

In .Net MicroFramework which is specifically designed for specialized hardware devices the Garbage collector the memory management is tied to the type system, making it possible to track memory at the object level. This allows the system to reclaim "dead" objects and removes this burden from the programmer. In addition, memory can be "compacted" to create contiguous free space for newly allocated objects, thereby avoiding out of memory conditions due to heap fragmentation. This is known as Garbage Collection (GC).

Every program uses resources of one sort or another—memory buffers, screen space, network connections, database resources, and so on. In fact, in an object-oriented environment, every type identifies some resource available for your program's use. To use any of these resources requires that memory be allocated to represent the type. The steps required to access a resource are as follows:

1. Allocate memory for the type that represents the resource.
2. Initialize the memory to set the initial state of the resource and to make the resource usable.
3. Use the resource by accessing the instance members of the type (repeat as necessary).
4. Tear down the state of the resource to clean up.
5. Free the memory.

Although the above paradigm seems very simple, freeing memory when it is no longer needed or attempting to use memory after it has been already freed are two important concerns. In .Net, the developer can use certain methods for explicitly cleaning memory.

***Multi-threading:***

.NET MicroFramework CLR provides multi-threading support, even when the underlying platform does not. While not a true multi-threaded kernel, the execution engine simulates one by offering time-sliced context switching using 20ms quantum. Threads are prioritized and interruptible (due to instruction level interpretation hardware).

## IV. Proposed work

We intend to provide hardware support to the .NET MicroFramework Common Language Runtime for embedded systems (CLR) on FPGAs so that programs targeted to run on CLR software engine are accelerated. The system is to be built atop the MicroBlaze softcore processor, which will run on a Xilinx FPGA board.

The very common solution to build any general VM is to have it as an interpreter and/or a JIT compiler or an AOT compiler. This kind of a software solution is relatively easier to build and cost effective, but it compromises on the performance. But the price they demand is flexibility. Moreover, complexity and cost are high for such an approach. Our objective is to exploit the advantages of both these approaches by employing both of them. In other words, we propose a co-design sort of an approach to achieve a better negotiation between cost and performance. Also, with this paradigm lies flexibility too. A goal of the co-designed CLR is to achieve better performance over a pure software solution. The hardware partition of the co-designed CLR is proposed to work in unison with the host processor, which can be a GPP (which in our case happens to be a softcore). The choice of an FPGA provides a development environment for easily shifting the partitioning between hardware and software to arrive at an optimized solution, and the flexibility to fit the design into targets of different sizes. Fig. 1 gives the overall block diagram of the system.

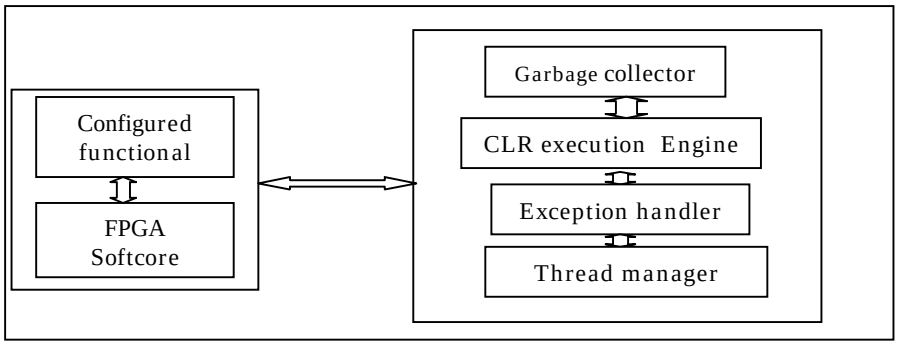


Fig 1: Overall block diagram

From a very high level, the following are the tasks to be performed by the CLR: Loading, verification, translation and interpretation, memory management and garbage collection, code management, exception handling and thread support. The hardware-software codesign approach is followed to design the system. All features that are beneficial to be hardwarized are configured in hardware. The software partition supports the remaining features and also provides system support required by the underlying hardware system. The instructions of the CLR are implemented in both the software and hardware partition. The hardware partition includes any instructions that can be implemented in hardware, like frequently occurring instructions such as stack manipulations, and arithmetic operations.

Our project will consist of the following modules: (1) CLR Execution engine (2) Garbage collector (3) Exception handler (4) Thread manager

(1) CLR Execution engine

A CLR Execution engine has been developed that maintains the stack in hardware[1]. The software executor, CLR engine, forms the software partition and is responsible for the execution of CIL code. The previous implementation[1] does not support object oriented and object modeling instructions that are necessary for implementing other features like garbage collection, threading etc. Our implementation enhances the existing engine with the object modeling instructions and other instructions that form part of the .NET MicroFramework.

Since the CLR engine is implemented using a hardware stack, handling stack overflows is of prime importance. The stack overflow problem is handled using a spill fill handler. The stack is maintained as a circular one. A backing store is setup as a linear block of memory to hold the spilt entries. Spilling of the stack is done when the stack becomes full. Instead of spilling the whole stack, half of it is spilt. A threshold is fixed for the filling of the stack. When the threshold is reached, the stack is refilled from the backing store. To exploit the concurrency of FPGAs and not to have the memory access overhead, dual port RAMs are used so that a read and write to the external memory can be done concurrently. This obviates the unnecessary stalls that might arise in the case of a single port RAM. Fig 2 shows a block diagram of the execution engine with the spill fill handler.

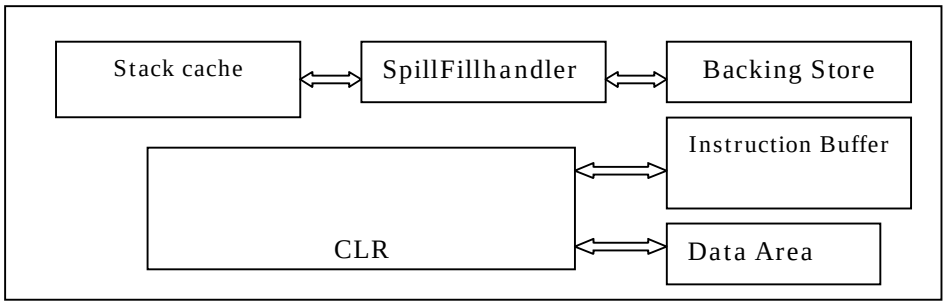


Fig2 : CLR Execution engine

(2) Thread manager

Thread management needs a lot of support from the hardware. It demands multiple set of special registers like Program counter and Stack pointer, and also multiple independent copies of the general purpose registers. Then, threading would involve keeping track in the processor, the thread currently in execution using a number of bits. There also needs to be a powerful scheduler to select the appropriate thread for execution, which in turn demands timer support from hardware. In our implementation, we use a MicroBlaze softcore processor which doesn't support multiple sets of registers. Thus, to implement threading atop the MicroBlaze, we will have to bear the cost of context switching. This is very expensive, with respect to both time and memory. Since it slows down the execution and also

demands precious memory (which is limited in an embedded device), it is not very efficient to implement it on the MicroBlaze. Thus, it can be implemented using a different underlying processor (a version that would support threading) or by configuring of the required special registers in the reconfigurable hardware.

(3) Garbage collector
Garbage collection used in .NET MicroFramework is the Mark and Sweep method explained in the previous section. The heap is be divided into fixed size pages. There is a Next Object Pointer which keeps track of the next available free page for the objects. Our design has a heap manager to control the marking and sweeping of the heap. A mark table is maintained which has information about marked pages. Fig 3 gives a high level design of the garbage collector module.

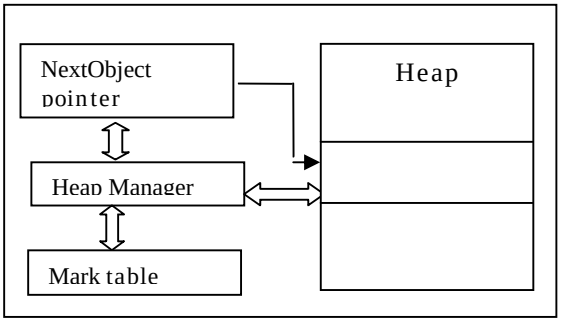


Fig 3 : Garbage collector

(4) Exception handler
Exception handling requires little support from the hardware, except for a few registers for exception identifier, handler address etc. Hence, the exception handler is handled in the software partition.

All these modules (components) together will constitute the .NET MicroFramework CLR.

## V. Conclusion

Many embedded devices are too small to support the full .NET Framework. Also, they don't need the entire functionality that the .NET Framework provides. The .NET Micro Framework provides a small, efficient implementation of the .NET runtime for smaller devices. Enhancing that with hardware support has great advantages to developers in terms of performance, speed and portability. Our implementation of the .NET MicroFramework CLR is be a hardware-software combination, comprising of the execution engine and the other constituents of the CLR (garbage collection, thread management, exception handling). The codesign approach lends enhanced performance to this hardware supported CLR as compared to the existing pure software implementation. The advantages and limitations of this approach are analyzed.

## VII. References

[1] Srinath S, Srinivasan.T, VidhyaBhushan.M, Ranjani Parthasarathi, Department of Computer Science, Anna University, 'Implementation of .NET CLR on FPGAs', 2005.
[2]Microsoft .NET Micro Framework white paper,
www.aboutnetmf.com/NET_Micro_Framework_Whitepaper_v_1.0.doc
[3] Article on 'Garbage Collection: Automatic memory management in the .NET Micro Framework' by Jeffery Ritcher, MSDN magazine, Nov 2000.
[4]David F.Bacon, Perry Cheng, David Grove, IBM T.J. Watson Research Center, 'Garbage collection for Embedded Systems'.
[5] L.V. Nagendra Kumar, International Institute of Information Technology, Gachibowli, Hyderabad, India, 'JVM Implementation in FPGAs', B.Tech final year Project report, 2002.
[6]The following article from the Intel website was referred.
"Itanium® Processor Family Performance Advantages: Register Stack Architecture", By Scott Townsend,
http://www.intel.com/cd/ids/developer/asmo-na/eng/affiliate/hp/20314.htm
[7] ECMA Draft (ECMA/TC39TG3/2000/3)- Part 3 IL Instruction Set, March 2000.
[8] Ryan Rakvic, Ed Grochowski, Bryan Black, Murali Annavaram, Trung Diep, and John P. Shen. Performance Advantage of the Register Stack in Intel® Itanium™ Processors**,** Microprocessor Research, Intel Labs (MRL),2002.