Implementation of the .NET CLR on FPGAs

Srinath.S, Srinivasan.T, Vidyabhushan.M, Ranjani Parthasarathi
{sridharan.srinath, sr.iniv.t, mailtovb} @ gmail.com, rp@cs.annauniv.edu
Department of Computer Science, College of Engineering, Guindy, Anna University, Chennai – 600025

Abstract - *Microsoft's .NET platform is a promising technology to achieve interoperability between programming languages, and true portability over different hardware and operating system platforms. 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. In this paper, we propose a design of a .NET embedded processor on FPGAs that improves the performance of the .NET Common Language Runtime (CLR).*

Keywords – *FPGAs, CLR, CIL, hardware software co-design.*

1. Introduction

With the proliferation of embedded devices, the system design paradigm has shifted from the conventional design to hardware-software co-design [11]. The focus is on building customized hardware and software. The hardware customization has been facilitated by Field Programmable Gate Arrays (FPGAs). 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. As a result, FPGAs are being deployed widely in the embedded market.

Software customization has been achieved using virtual machines (VM), in particular using the JVM framework. However, the potential of the .NET technology in the embedded segment, is yet to be explored. While varied implementations of JVM on hardware exist in the embedded domain, no known support has been provided for .NET in embedded architectures - either through CLR implementations or through JIT compilation into native code. This is due to the fact that while JVM is small in size, and amenable to hardware implementation, .NET is not so. Hence, in this paper, we explore the ways in which one can exploit the flexibility of FPGAs to accelerate the programs targeted to run on .NET virtual machine (CLR), by considering a constricted set of the .NET instruction set architecture (ISA).

The paper is organized as follows. Section II discusses the past trend regarding the design of virtual machines, and also about the approaches that are taken. Section III introduces the CLR and shows how different it is from the JVM. Section IV talks about our scheme of implementation of the CLR highlighting the design decisions.

II. Background

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 [7].

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. The intermediate form acts as a means of communication between the high level front end and the low level back end. 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, add-top-two etc.). Thus, it becomes desirable to model VMs as abstract stack machines.

Although the VM system solves the problem of portability and achieves its objective of "write once, run anywhere", the translation of the intermediate form to its native form is an overhead and hinders the faster execution of programs written for the VM system. Another problem with such an implementation is that the programs written for such a system cannot take advantage of the special features of the underlying architecture, which might accelerate the execution of programs.

With the increase in demand of homogeneous computing, which reduces the development time, the use of virtual machines have become more prominent despite their inferior performance and some of their disadvantages.

There are many ways of overcoming the slower performance of the virtual machine environments, some being hardware solutions and some being software solutions. The following sub-section presents the methods with respect to the Java system.

The Internet revolution & the advent of Java:

In the late 1990s Sun Microsystems released their Java [8] language system. This system, again, was based on an abstract stack machine. The wide availability of the Java system, coupled with the rich set of libraries it provides, has proliferated the use of JVM. The intermediate code form for

Java is "Byte Codes" and the execution environment is the Java Virtual Machine (JVM).

Many advances have been made in order to have a faster execution environment. One way to achieve a faster runtime is by having dedicated Java Coprocessors, which can help the host processor execute Java code better, faster and in an efficient manner. Some examples are JVXtreme, JStar and Xpresso [6]. Embedded systems and real time systems, which execute Java Code, adopt this kind of a solution. Although this idea seems to be attractive, we still need to have a GPP running on the target machine. Moreover, this increases the cost too.

The other extreme is to have specific processors, which can directly execute Java Bytecodes. These processors are called Java Processors [5]. The beauty of these processors is that the very ISA of them is the JVM ISA, meaning that they work on bytecodes directly and run them directly on hardware, thereby achieving speedier execution. Moreover we are done away with the additional and time-consuming tasks of interpretation and JIT compilation. Thus the performance benefits that can be reaped from such an approach are many compared to coprocessor technology.

Another way to speed Java execution on a standard RISC architecture is to extend the architecture itself, to directly execute Java instructions. ARM designers added a new Java instruction set to the classic ARM architecture [5]. The Java ISA is executed in a Java mode, which is entered on a branch. In the Java mode, the CPU executes Java byte code instructions.

There have also been complete FPGA implementations of JVM. One such work is JVM implementation on FPGAs [1], which draws its inspiration from Sun's Pico Java Processor. In this work, a Java microprocessor core using FPGA technology is experimented. The processor core natively executes Java bytecodes defined by the Java virtual machine. It eliminates the need for commonly used interpreters, JIT compilers and their overhead. The core accelerates the Java virtual machine runtime environment. It executes the most commonly used instructions in hardware. Complex instructions are micro coded, with the most complex ones trapped and emulated in software. The advantage of the above-mentioned approach is that it increases the speed by 5 to 10 times than the conventional the JVM implemented in software.

So, it is evident that the JVM has been given much attention by the embedded systems group and reconfigurable computing groups. .NET on the other hand, is an upcoming technology and not so much work has been done as regards its embedded development. Although a chip that is capable of running a very restricted subset of the CLR is already out in the markets, its scope of application is pretty narrow. We intend to build a system which can have a broader application domain but with the features that might typically be required by an embedded system.

III. The .NET environment and the Common Language Runtime

It was only so recently, in the mid-2000, did Microsoft reveal a new technology called the .NET. Microsoft claims that "the .NET Framework is a new computing platform that simplifies application development in the highly distributed environment of the Internet" [2].

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). We prefer to use the name of CIL. More information about the CLR can be had from [7] and about CIL from [3].

The CLR and the JVM:

Although the CLR and JVM are both stack-based architectures and share a few common features, there are a few vital differences between them. A few of those differences are presented here.

First of all, the JVM is so designed to run only Java code efficiently. Although it's quite possible to make the JVM coexist with other languages, the JVM is necessarily a sub-optimal multi-language platform [9,10]**;** whereas, the .NET framework and the CLR has been designed from the ground up having language interoperability in mind. The assembly language of the CLR is an object-oriented assembly and includes generic instructions, but the JVM ISA has got no generic instructions. Another difference between the JVM and the CLR is that Java doesn't provide provisions for writing native code or type-unsafe features of typical programming languages (native pointers etc). But the .NET framework differentiates between managed and unmanaged code. The unmanaged code is capable of executing code, which can run out of the vigilance of the CLR. .NET adheres to the Virtual Object System (VOS). Value types can only be primitives in Java, where as the .NET supports **`struct`**s and **`union`**s. CLR includes provisions for automatic boxing and unboxing, whereas JVM doesn't. The parameter passing conventions differ in Java and .NET.

Thus the various design issues and parameters are quite different for the JVM and the CLR. Those that apply for the JVM do not apply for the CLR.

A brief description of the CLR architecture:

The CLR maintains an **instruction pointer(IP)** to hold the address of the next instruction to be executed. The **evaluation stack** is used to contain intermediate results of computations. Local variables are held in a **local-variable array** and the incoming arguments are held in an **argument array.** Besides, the following too are a part of the CLR: A **local memory pool** for dynamic allocation of objects, **method info handle** to contain the information about the currently

executing method and a **return state handle**, which is used to restore the state of a method on return from a callee method. The CLR gives high priority to security issues and hence has a **security descriptor** to record security overrides.

IV. Proposed Work

We intend to implement the .NET Virtual Machine i.e. the Common Language Runtime for embedded systems (CLR) on FPGAs so that programs targeted to run on CLR software engine are accelerated. As the design space is restricted, we take into account only those features of the CLR that are indispensable in an embedded system. The system is to be built atop the MicroBlaze softcore processor [12], 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. Possible hardware implementations include techniques such as native processors or coprocessors as had been discussed above for the case of Java. 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. The Fig. 1 gives the overall block diagram of the system.

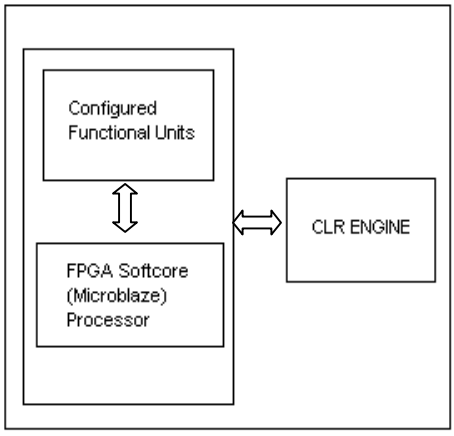

Fig. 1 System Diagram for .NET processor

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, thread support. High-level tasks such as class loading, security checks, verification, exception management subsystem, garbage collection etc are implemented in the software partition. The software partition also provides system support required by the underlying hardware system. All the instructions of the CLR are implemented in the software partition. It is desirable to have the hardware partition include any instructions that can be implemented in hardware. It includes the conventional instructions such as constant operations, stack manipulations, and arithmetic operations.

The CLR Engine:

The software executor, CLR engine, forms the software partition and is responsible for the execution of CIL code. Given below in Fig. 2 is a high-level block diagram of the CLR engine.

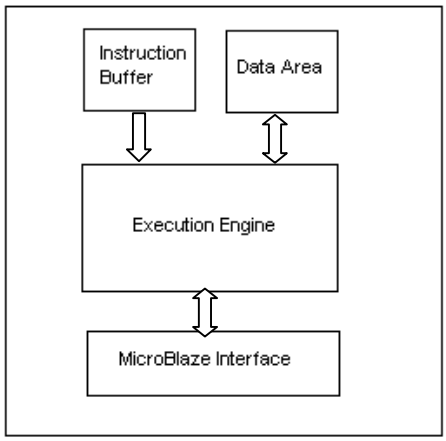

Fig. 2 A conceptual view of CLR engine

The CLR engine unit is divided into four main parts: **the MicroBlaze interface, execution engine, the instruction buffer, and the data area**. The instructions are first loaded into the instruction buffer (the PE file) to the engine through the host processor interface. The instructions reside in the instruction buffer, from which the execution engine reads and executes them. The data area is used to hold the stack and maintain the local heap from which dynamic allocations can be made. The MicroBlaze interfaces with external peripherals.

Method:

This section explains the working of the CLR engine. After getting invoked, the CLR engine first loads the assembly (a standalone executable, similar to class file in Java) into the ***instruction buffer.*** It is worth noting that we are running on a memory-restricted environment, in which we don't have the provision of loading very large assemblies. The memory needed, is necessarily a bottleneck.

After the loading process is finished, control is handed over to the code verifier. The loaded IL bytecode is analysed to ensure that all relevant correctness and security conditions are satisfied.

After the code-check phase, the process of actual interpretation starts. The CLR engine's core can be roughly portrayed by a large switch-case running in an infinite loop, as shown in the figure given below.

```
while(moreInstructions()){
      switch(next_instruction())
      {
            case ADD:
                  doAdd();
                  break;
            .
            .
            case PUSH:
                  doPush();
                  break;

      }
}
```

The code written above is very similar to a software VM. We do not employ more sophisticated techniques such as JIT or AOT because a few of the instructions are to be implemented directly in hardware. This is explained below.

In such an implementation, the functions that get invoked for getting the work done contains code to accomplish what is expected of the invoking instruction. For example, the add instruction of the CLI can be implemented as follows. Pseudo codes of pop and add are given below.

```
void pop()
{
    SP = SP – 1;
}

void doAdd()
{
    if( typeof TOS is 'int' ){
    int i1 = stackTop();pop();
    int i2 = stackTop();pop();
    push(i1+i2);
}
```

Our approach:

The novelty of our approach lies in how we implement the hardware part of the system in FPGAs. We make full use of all the features of field programmable logic.

We partition our work into two major modules. The first part would involve the porting of the CLR onto the MicroBlaze softcore, which has already been explained. The second part will be the hybridisation of the softcore. It is in the second part that we identify those instructions that can possibly lend themselves to parallelization (most of the base instructions of the CIL) and implement them directly on hardware, thus accelerating them. The doAdd() function given above will make an exquisite example. The two stack pops can be done in parallel, thereby halving the time required for its execution.

Also, frequently used instructions in the .NET ISA are found, separated out from the other set and they too are implemented in hardware, so that there can be a performance improvement.

This process of hybridisation involves reconfiguration of the underlying soft core so that we can have dedicated hardware units for our custom functions. Reconfiguration can be done in two ways:

i. Modification of the existing MicroBlaze architecture itself
ii. Augmenting the softcore with new functionalities.

These custom functions include all the parallelizable and frequently used instructions.

When the execution engine encounters an instruction that has been implemented in hardware, it transfers control to the hardware implementation, so that software emulation is bypassed.

We wish to illustrate the performance gain that can be achieved by such hybridisation through the following example case.

The MicroBlaze softcore contains 32 32-bit registers. A few of these registers are employed for internal purposes. For instance, we have one of them as IP, one as stack pointer (SP), and one as flags register for internal usage of the system. The topmost two entries of the stack will always reside in two registers. These two registers are called TE1 and TE2 (Top element). All stack manipulations should take into account these two registers and make sure that they reflect the stack's topmost two elements. For example, when a push is made onto the stack, the SP register is incremented, TE2 is written onto the memory location whose address will be (SP-4), TE2 is replaced with contents of TE1 and TE1 is overwritten with the new value. All these are done in parallel, so that we don't have an unnecessary overhead in stack manipulation.

The figures given below (fig 3 and fig 4) depict the real and conceptual scenarios.

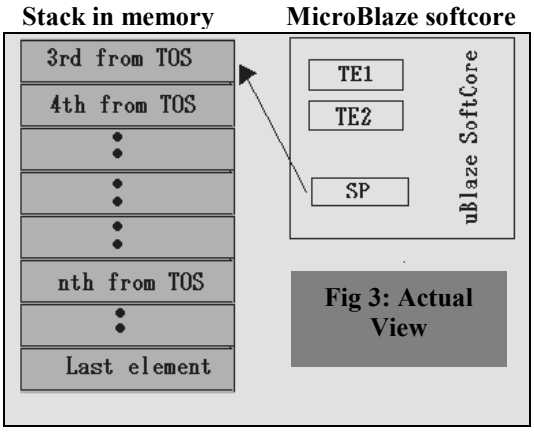

Fig 3: Actual View

It should be noted that not all instructions need two operands to operate on.

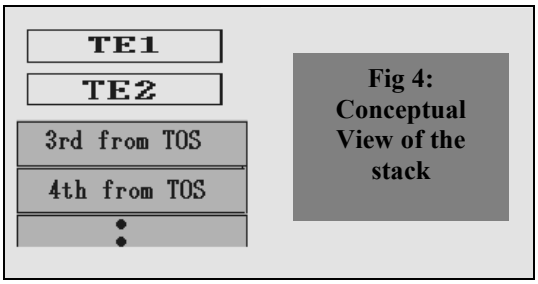

Fig 4: Conceptual View of the stack

Now, because we have the topmost entries inside the register, all of the quick instructions such as the arithmetic instructions or primitive instructions can be accomplished in two clock cycles as compared to 7 clock cycles in a software implementation. Consider the following example implementation for **`add`** instruction of the CLI.

Example implementation:

We now explain in this section one way of implementing the add instruction. The add instruction pops the topmost two values of the stack, adds them and pushes the sum onto the stack. Algorithmically, the add instruction can be given as

```
void doAdd()
{
    if( typeof TOS is 'int' ){
    int i1 = stackTop();pop();
    int i2 = stackTop();pop();
    push(i1+i2);
}
```

A dedicated hardware can be designed to have this operation completed in two clock cycles. The unit has a built-in temporary register, say TEMP, for internal usage. With such a hardware unit in hand, we can accomplish the addition in two cycles thus:

Cycle I:

TE1 ← TE1 + TE2;
TEMP ← mem [SP];

Cycle II:

TE2 ← TEMP;
SP ← SP – word_size

This example clearly illustrates how hardware implementation can bring down the number of clock ticks required to accomplish an operation. All arithmetic and bitwise operations can be implemented similarly and execute faster.

To exploit the concurrency of FPGAs and not to have the memory access overhead, we can have dual port RAMs 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.

Moreover, it is also intended to implement in hardware, those instructions that are more frequently encountered than others. Doing so, according to Amdahl's law, we can reap performance benefits. A typical example would be to have a dedicated exception handling unit and thereby having **`throw`** and **`catch`** instructions in hardware. Other instructions that do not lend themselves to such optimization are emulated in software.

V. Possible extensions

One point to be noted is that the above implementation will need an RTOS for its functioning, which can be considered as an unnecessary memory overhead (the kernel etc.). The design can be extended to run without any RTOS, thereby saving the memory required for the kernel. The memory can be reused to accommodate more modules and features.

Another possible extension to our work will be to implement the entire system of CLR on raw FPGAs, although it involves a lot of complexity. The design can be cast in a die to have a programmable .NET processor at the end.

VI. Conclusion

Although technologies like Java and .NET become predominant day by day due to the increase in demand for homogeneous computing, their performance, due to interpretation, becomes inferior. It, therefore, becomes imperative to improve the performance. The .NET technology is a more desirable solution than Java because of its language interoperability. In this work, we have discussed methods to improve the runtime performance of the .NET system.

VII. References

[1] L.V. Nagendra Kumar, International Institute of Information Technology, Gachibowli, Hyderabad, India, 'JVM Implementation in FPGAs', B.Tech final year Project report, 2002.

[2] Plataforma .NET - available at 'http://people.ac.upc.edu/enric/PFC/Plataforma.NET/p.net.html', 2002 *Topics Referred in this URL - Objectives and Future Projects.*

[3] ECMA Draft (ECMA/TC39TG3/2000/3)- Part 3 IL Instruction Set, March 2000.

[4] K John Gough, 'Stacking them up: a Comparison of Virtual Machines', In the proceedings ACSAC-2001.

[5] Java Processors – available at www.elecdesign.com/Articles/ArticleID/3500/3500.html

[6] An introduction to the Pico Java processor - Available at : http://www.pages.drexel.edu/~ooo22/KImages/picoRep.htm, 2002.

[7] Erik Meijer and Jim Miller, 'Technical Overview of the Common Language Runtime' Microsoft, 2001

[8] Java Virtual Machine – available at http://java.sun.com

[9] K John Gough, 'Parameter Passing for the Java Virtual Machine' Australian Computer Science Conference ACSC2000, Canberra, February 2000, IEEE Press.

[10] K John Gough and Diane Corney, 'Evaluating the Java Virtual Machine as a target for Languages Other than Java' Joint Modula Languages Conference JMLC2000, Zurich, September 2000

[11]Hejun Ma, Ken Kent, David Luke, 'An Implementation of the Hardware Partition in a Software/Hardware Co-Designed Java Virtual Machine', In the proceedings of IEEE on May 2004,Volume 4, ISSN: 0840-7789 ISBN: 0-7803-8253-6.

[12] www.xilinx.com/xup/mb_ref_guide.pdf, Xilinx Inc.