FFast : Efficient Application of Compiled Simulation Techniques to a Fast ISS Over a Virtual Machine

Trouvé, Antoine
Institute of Systems, Information Technologies and Nanotechnologies

Murakami, Kazuaki
Kyushu University

FFast: Efficient Application of Compiled Simulation Techniques to a Fast ISS Over a Virtual Machine

Antoine Trouvé †‡, Kazuaki Murakami †‡
† Institute of Systems, Information Technologies and Nanotechnologies (ISIT), Japan
‡ Kyushu University, Japan

***Abstract*—This paper presents FFast, a compiled ISS (instruction set level simulator) written in C# for maintainability and portability reasons. If compiled ISS methods are well studied, this work presents an original implementation mixing compiled and interpreted approaches. The presence of the .NET's virtual machine between FFast and the host machine introduces some overhead which are also studied. In this context, the main contribution of this paper is to propose a method to implement efficiently an ISS over a virtual machine. Performances are assessed using a small benchmark as a proof of concept. Experiments show that a complete ISS written in C# can reach more than 200 million of instructions per seconds on an Intel Core2 Duo Extreme processor at 2.6 GHz.**

***Keywords*-ISS, Compiled Simulation, Virtual Machine, C#**

I. Introduction

If VMs (virtual machines) come with overheads in term of execution speed and memory usage, they do provide with an abstraction of the host machine which enables end-user application developers to forget about constraints due to the presence of the host machine. Therefore, they can concentrate on what matters: the implementation of the very algorithms or user interfaces. Programs targeted to VMs are compiled into v-code (virtual machine code) independent of the host processor; the same files can be executed on many platforms without needing any recompilation (provided that the VM is ported to the given host). In all popular high-level VMs, the v-code is dynamically translated into native code (or n-code): it is JIT (just-in-time) compilation. This method enables to perform some run-time optimisations on the code based on profiling data. The memory management is also usually abstracted, through garbage collection for instance.

The idea of using VM between the user applications and the host machine is almost as old as software engineering: programs written in the Smalltalk [1] and the Self [2] languages are executed above full-featured VMs since 20 and 30 years respectively. Sun's JAVA and Microsoft's .NET are two examples of recent successful language framework based on a high-level VM with JIT compilation and garbage collection. Both use a stack-based v-code (also called byte-code because of the size of instructions' op-code) for easy instruction decoding.

In the .NET framework, programs (many languages are supported) are compiled into a v-code (bytecode) called CIL (Common Intermediate Language). Then, when the application is executed, this v-code is JIT compiled into n-code by the VM and executed on the host processor. There are two .NET VMs available: CLR (Common Language Runtime) [3], developed by Microsoft, does only run on Windows; Mono [4] is an open source, free .NET VM supported by Novel which can be executed on Linux, BSD, UNIX, Mac OS X, Solaris and Windows.

If very popular in the high-end application industry, C# (like Java) is not used at all when performances become a key factor, like instruction set simulator (ISS). In effect, it has the reputation to be slow and resource hungry when compared to other n-compiled languages like C or C++. It is however not the point of view defended in [5] which presents a system level modelling and simulation environment based on .NET which claims having less than 10% of execution speed penalty comparing to System-C.

This paper introduces FFast, an ISS (instruction set level simulator) implemented in C#, which takes advantage of the substantial productivity enhancement provided by this language at the cost of small performance penalties. One of the goal of this paper is to show that high-productivity languages like C# can be as well used when performances matter, like in ISS development.

II. Related work: simulation techniques

During the design process of embedded processors, ISS (instruction set level simulators) are powerful and flexible tools for fast design space exploration. Main qualities of an ISS are, by order of relevance:

1) *its execution speed.* It is usually expressed in million of instructions executed per second (MIPS),
2) *its traceability.* It corresponds to the ability of the simulator to report its own activity,
3) *its re-targetability.* It is the ability that has an ISS platform to support easily a new target architecture,
4) *its portability.* It is the ability for a simulator to be executed on several host architectures.

FFast, the simulator presented in this paper, performs well is those four domains, thanks to the use of compiled ISS techniques (1) and C# (2,3,4).

The traditional approach to implement an ISS (called *interpretative simulation*) implies fetching instructions each time they have to be executed. Simplescalar [6] is a mature flexible and re-targetable interpretative simulator which supports a wide range of off-the-shelf processor architectures thanks to an active community. It provides with an execution speed

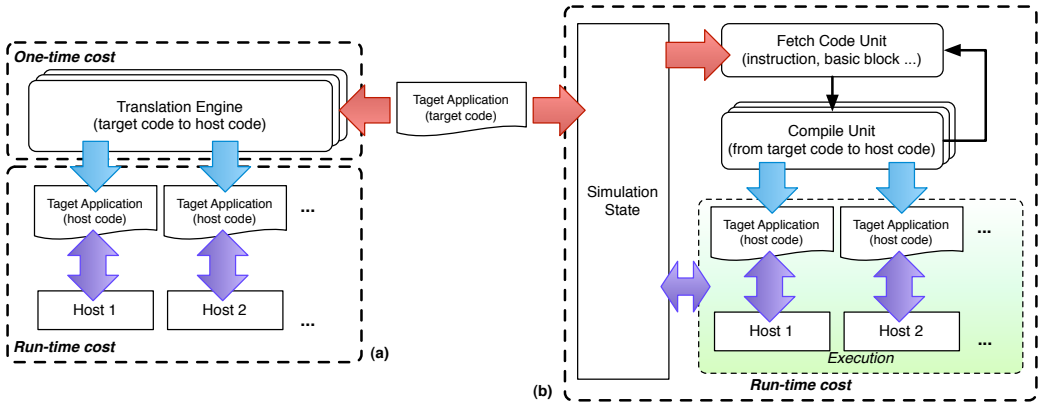

Fig. 1. Compiled Simulation Flows: (a) static (b) dynamic

acceptable for average sized target application: our tests show that it is about 130 times slower than native hardware in its fastest version (`sim-fast`) on an Intel Core2 Duo. It is however from 2 to 4 times slower than our ISS which uses less mature code but faster algorithms.

The usual technique to speed up simulation is to compile a part or all of the simulation process for the host. This method takes its roots in research on fast RTL simulation. We use the term C-ISS (compiled ISS). Algorithms used in C-ISS resemble the one used in VM. For instance [7] assesses methods to optimise the Smalltalk-80 VM; all problematics it discusses do also matter in the domain of C-ISS.

Figure 1 (a) illustrates the typical flow for the static C-ISS approach. Simulation is done in two passes. The target program is first translated into n-code, typically by direct binary translation ([8], [9]), using techniques which resemble macro-expansion. The generated program is a stand-alone, fast ISS of the target application on the target architecture which can be executed as many times as needed on a given host machine. The main drawback of such method is its obvious poor portability (n-code generation is host-dependant), which has been partially addressed in [10] (at the cost of performances) by using C as an intermediary language to generate binaries in host native code. [11] presents a static C-ISS technique which also uses C as intermediate for native code generation. It however addresses the execution speed issue through a low-level code generation interface which aggressively uses host machine resources, and reaches speed close to the hardware speed. If very fast, such an approach does limit the portability of the ISS since the code generation units have to know how to use the host machine resources efficiently. Portability is one of the strongest point of FFast since the host is always the same: .NET's VM (but performances can vary between different implementations of the VM)

Static C-ISS do have limitations. The first one is their limited support of self-modifying programs (which is the case of OS with dynamic linking). The second one is the impossibility to lead dynamic optimisations to take advantage of the dynamic state of the host.

Those limitations can be addressed through the dynamic C-ISS technique which aims at bringing the flexibility of interpretative ISS to C-ISS. Figure 1 (b) shows the typical flow of a dynamic C-ISS. At run-time, instructions of the target program are fetched one by one (like does interpretative ISS) or by block (typically basic blocks). Those are then JIT compiled into n-code before being executed. The overhead due to multiple calls of an external compiler is then expected to be amortised in loops. It can also be reduced through a pre-compilation phase before running the simulation itself.

However, due to the overhead of the JIT compilation step, this algorithm is usually not applicable in state. [12] uses a cache of already compiled instructions to reduce the overall compilation time. To support self-modifying code, the cache is updated if the program memory is written at run time. The resulting ISS is however only 2 times faster than Simplescalar. [13] proposes to add a pre-compilation step executed before each run, which generates target program specific templates to make it possible for the ISS to compile instructions on the fly very quickly. Like in [12], templates are updated when needed in the case of self-modifying code. From the same authors, [14] choses to generate a decoder specific to the program in an algorithm the authors called *hybrid instruction set compiled simulation*. This method achieves 3.5 times speedup comparing to Simplecalar.

FFast is a static hybrid compiled / interpretative ISS implemented in C# which implements a semi-automatic C-ISS approach. It does not support self-modifying code, but it uses JIT compilation technique to take advantage of profiling informations at run time. Furthermore, it is more flexible than previous woks in two ways. (1) It is also able to switch rapidly from CISS mode to interpretative mode for instructions or resources which can not be compiled (for maintainability reason). (2) Instead of C or assembly, it uses as intermediate representations C# then CIL which behave the same on all .NET VMs.

[15] also presents an ISS called *SyntSim* which mixes static C-ISS and interpretative ISS; it is almost two times faster than FFast. It also introduces several optimisation techniques to ac-

celerate its C-ISS engine, all of them being also implemented into FFast. However, it still uses C as intermediate language.

In this context, this paper assesses for the first time of our knowledge new issues which come with the introduction of a VM beneath an ISS, as well workarounds which can be implemented.

III. The FFast ISS

A. Structure of FFast

FFast is based on an interpretative ISS (which will be called I-FFast) implemented in C# for productivity reasons. It was designed from the ground to be flexible more than fast to allow easy design space exploration. Another requirement was its portability in an heterogeneous research environment.

In I-FFast, architectures are described through resource and ISA (instruction set architecture) description. Resources correspond to datapaths and memories which can be explicitly used by the ISA. They are described through a tree (structural approach) of atoms (behavioural approaches).

This description approach is similar to the one used in the LISA [16] or the nML [17] languages and allow to support very complex architecture. However, resources and ISA descriptions are passed to I-FFast by minimalist ”glue classes” written in C# instead of text files for maturity reasons.

Resources' behaviour can be easily monitored using the events of their corresponding object instances (see [18] for a description of the event paradigm in C#). Events are triggered by resources with specific actions (read, write or execute), and executed behaviour is the one registered by an observer object. For instance, to trace executed instructions one has to register a function on the ”execute” event of the processor resource. This approach enables a high level of traceability at a very low engineering cost (the language deals with the complexity under the hood)

B. Implementation of the C-ISS

FFast is an evolution of I-FFast which leverage speed through dynamic C-ISS techniques as explained in the related work. Figure 1 (b) presented the general flow for such ISS. In the case of FFast, compiled code units are eBB (extended basic blocks, i.e. they can contain some branches under certain conditions) and the host is unique: .NET's VM (CLR or Mono). As a consequence of this last point, only one implementation of the compile unit is needed for all host systems, the portability work being deported to the VM (which is considered as a zero cost here).

The compile unit is made of two steps: gathering and compilation. Starting from a given PC (the starting PC ”sPC”), the gathering algorithm integrates in a ”sequence” all reachable instructions. It uses a recursive method. For all instructions already in the sequence (initially the one at sPC), their successors are also integrated. The ”successors” term refers to all instructions for which it can be determined statically that they can be executed after a given previous instruction. As a first approach, only basic blocks are gathered so branches do not have any successors. However, to accelerate some simple patterns like ”switch-case” and small loops, (1) conditional jumps have one successor which corresponds to the branch not taken (2) branches which target (if known during sequence gathering) is already included into the sequence being gathered are also integrated (sequences are not simple basic blocks anymore). Therefore, all sequences end with a jump instruction or a ”exit” system call (which ends the simulation).

Gathered sequences are translated into C# using an algorithm which almost boils down to the macro-expansion of instructions. To serve that purpose, resources are able to emit code corresponding to their three basic behaviours (read, write execution). In generated C# code, each instruction is preceded by a label corresponding to its PC to allow branches inside a sequence. Static branches are implemented using `goto` if the target is inside the same sequence, or a `return` otherwise; dynamic branches are always implemented with a `return`. Resources' code emission functions also inline routines which trigger events to allow the same level of traceability as I-FFast in FFast. It however leads to performance penalty which will be evaluated later.

After translation, obtained C# statements are compiled using the CodeDom library provided by .NET, the result being a class (one per sequence) with a public execution method (a ”ff-class”). This ff-class can then be loaded dynamically by the VM and executed. This possibility offered by C# saves the effort of integrating this ISS into a re-targetable compiler (like [12]). Generated ff-classes are cached and indexed by their starting PC. Therefore, each time FFast reaches a PC (outside a sequence), it checks if a sequence starting with this PC is in the cache; if not it goes through the gathering and compilation steps before executing the corresponding ff-class. It is the sequence selection phase. This method supposes that the instruction memory is invariant during the program execution; this hypothesis is ensured at run-time by prohibiting write operations in the instruction memory during the simulation. This constraint does prevent (from now) the execution of most operating systems on FFast.

Through this approach, FFast is expected to run faster than I-FFast thanks to the removal of instruction fetching during execution, the execution of the behaviour of the simulation without function calls (flattened behaviour), the precomputation of all static branches and the integration of some static branches.

C. Performances

This section assesses the performances of FFast using the MIPS instruction set. As a test-bench, two kernels extracted from MiBench are used: CRC32 and Blowfish.

Performances on an Intel Core2Duo at 2.6 GHz without and with statistic gathering, are shown on Table I. The speed is expressed in MIPS (million of instructions per second) and slowdown compared to hardware speed (in brackets). Numbers are obtained through the linear extrapolation of 7 runs of both algorithms with different input sizes. This table also emphasises the different profiles of the algorithms: CRC32

is very small with small eBB and Blowfish is larger with mix-sized eBB.

TABLE I
SPEED OF FFAST WITH CRC32 AND BLOWFISH.

Algorithm	CRC32	Blowfish
Static Properties of Programs		
Number of eBB	13	67
Number of hot eBB	2	9
Average size of eBB	7.5 insts.	17.86 insts.
Simulation Speed in MIPS (Slowdown Comparing to Hw. Speed)		
I-FFast	0.13 (20508)	0.12 (22217)
FFast	1.2 (2222)	1.7 (1565)
FFast with stats	1 (2694)	1.3 (2028)

FFast runs in average at respectively 1.5 and 1.1 MIPS with and without statistic gathering. It is 10 times faster than I-FFast, but more than 1500 times slower than hardware speed, i.e one target instruction is translated into 1500 host instructions in average: the simulation overhead remains significant and above related works.

The overhead due to the VM's JIT compilation is negligible comparing to overall execution time, but other phenomenas constitute performance bottlenecks.

First, if the size of the eBB is roughly linearly correlated with the the G/C (gathering/collection) time, the slope is so small that the evolution is not significant (≤ 0.03 ms per target instruction). This overhead has however an important constant component: 100 ms per sequence if the system is warm, up to several seconds otherwise (effect of assemblies loading). For instance, a sequence of less than 20 target instructions (without loops) would take less than 10 ms to execute, but more than 100 ms to compile.

Secondly, frequent calls to the C# compiler leads to the call of the garbage collector (GC) which execution time is linearly correlated to the size of the inputs data of the program (from 0.01 to 0.02 ms per input byte according to our experiments). Considering the fact it makes the system swap, the GC overhead can become significant with inputs of several mega-bytes (up to 99% of the total execution time). Calls of the GC are typically located after the execution of inner data processing loops.

Lastly, other overheads due to the use of C-ISS techniques include the selection phase. Our experiments (not detailed here) show that when the size of the simulation increases, more than half of the extra overall simulation time (without GC) is due to the selection phase itself: it overpasses the execution time of the sequences.

IV. ACCELERATE FFAST

A. Spend more time in C-ISS mode: Branches Integration

The previous section has confirmed that using C-ISS techniques can speedup the execution of more than one order in FFast. It has also enlightened that the number of generated sequences is significant in the global execution speed and entails the overhead of: (1) the gathering/compilation and selection phases (2) the garbage collection.

In this context, it is desirable to reduce the number of sequences. To achieve this goal, the algorithm presented in this section will try to include as many branches as possible in sequences during the gathering phase: it is the "allb" method (for "all branches").

Depending on the type of branch, different strategies have to be adopted. Static branches' behaviour is known before their execution: during the gathering, their target can be added to their list of successors. This method is known as "branch pre-computation".

Dynamic branches can not be processed using the same method as their target is not known statically. This issue is overcome through two steps: (i) statistics collection (ii) re-compilation. Step (i) is done during the execution of the simulation: each time a dynamic branch is taken, its target is added to a list of known successors. Then, if this target is not in the sequence, it exits (using a `return`) and generates a "miss". If a given sequence generates too many misses, it is uncached and re-compiled using the new known successor list (step ii) as soon as its starting PC is met again. C# code generation for dynamic branches has to be updated in two ways: (1) for each known target, if it is included into the same sequence a `goto` is generated, a branch miss is raised otherwise (`return`) (2) each time a branch is taken, its list of known successors is updated.

To reduce the number of sequence misses at the beginning of the simulation, a pre-process is done before the simulation which aims at detecting all the function calls and updating in consequence the corresponding return instructions' known target lists (those are dynamic branches). It is the "allb-h" method ("h" for "heuristic"). During tests (not detailed here), FFast/allb-h has shown to be in average 25% faster than FFast/allb (1000% in some extreme cases)

Results (shown later in Figure 2) shows that FFast/allb-h is respectively 81 times and 12 times faster than FFast on CRC32 and Blowfish using the same modus operandi as in section III-C. This speedup is mainly due to the almost total deletion of the G/C and GC overheads. In the other hand, as sequences are getting larger, the JIT overhead increases. If this overhead is constant, it can not be neglected for small runs as shown in II. It shows the ratio JIT time / total simulation time for three runs of blowfish with FFast/allb-h. Even with inputs of 100Kb, this overhead counts for more than half the execution time. This discussion does not apply to CRC32 which sources are small enough to make JIT compilation time negligible ($\leq$ 1 % for similar input size). FFFast is an optimised version of FFast/allb-h described in the next section which enables to greatly reduce JIT constant overhead.

B. Reduce JIT overhead: Function level compilation

FFast/allb (and allb-h) does enable to build very large sequences (potentially the whole programs), which would be translated into one unique large C# function. If this technique can dramatically accelerate the simulation, it has some negative impact due to two mechanisms: (1) compilation time can get long for very large C# functions (2) very large compiled

TABLE II
JIT OVERHEAD COMPARING TO TOTAL SIMULATION TIME IN BLOWFISH.

Input Size (bytes)		*1000*	*10000*	*100000*
# instructions		592280	2224655	36685505
FFast/allb-h	JIT time	$\simeq$ 6010 ms		
	JIT overhead	70.5%	62.5 %	55%
FFFast	JIT time	$\simeq$ 1850 ms		
	JIT overhead	40%	35 %	30%

units may make the VM manipulate large objects which would cause the system to swap and caches to be saturated.

A way to reduce the size of JIT units is to fragment generated C# code according to the structure of the target program. Using the function detection algorithm implemented in FFast/allb-h, the sequence translation algorithm has been added the capacity to segment the generated code into several methods (one per actual detected function in the target application). This approach is however only efficient if used with allb-h which makes sure that caller and callee are integrated into the same sequences. It will be called "FFFast" (for Function-FFast). To limit the overhead of function calls during execution, small functions ($\leq$ 10 instructions) are also inlined.

Table II shows that by using FFFast, the JIT overhead can be divided by 3.25 in the case of Blowfish.

C. Analysis

This section will assess the performances of FFast and FFFast using the same conditions as in section III-C.

TABLE III
SIMULATION CONSTANT OVERHEAD

Algo.	FFast	allb	allb-h	FFFast
	Number Of Sequences			
CRC32	13	3	1	1
Blowfish	67	8	1	1
	Constant Overhead (ms)			
CRC32	$\sim$ 1500	449.4	313.53	344.00
Blowfish	$\sim$ 6000	2747.23	641.33	801.00

Table III shows for each algorithm and kernel the number of generated sequences and the constant overhead due to sequence gathering and compilation (it ignores JIT overhead, already shown in Table II). Due to the GC overhead exposed in section III-C, G/C and selection time can not be determined precisely for FFast.

The number of generated sequences drops significantly from FFast to FFast/allb to end up at 1 with allb-h as predicted: in compiled C programs most dynamic jumps correspond to `return` statements (or `longjumps`, not present in the benchmark). It leads to a proportional diminution of the constant overhead as wished. It should be noticed that the gathering/compilation time slightly raises with the use of FFFast: it is a natural consequence of the complexification of the C# emission algorithm.

The graph in Figure 2 shows the slope of the linear regression of the functions $speed = f(\#\ target\ instructions)$ (the speed in linear regime, SLR), which is the speed of the simulation for infinitely long run.

The benefit of FFast/allb-h over FFast is obvious. The SLR is multiplied by 80 and 10 for CRC32 and Blowfish respectively. Code segmentation in FFFast, which aimed at diminishing the constant overhead due to JIT compilation, is also effective: results (not detailed here) show that for small runs, the execution time is about one fifth of the one of FFast/allb-h. Surprisingly the SLR of FFFast is also greater that the one of FFast/allb-h (around 20 MIPS) which shows that the overhead due to extra function calls is not significant.

This speed can be almost doubled if array bounds checks (triggered at every memory accesses) are turned off in the ff-class through the C# keyword "unsafe": FFast/allb-h reaches 225 MIPS for CRC32 and 80 MIPS for Blowfish.

The simulation speed of Blowfish is generally lower than CRC32's one. It is entirely due to the execution time of the unique sequence (no G/C, GC nor JIT overhead). Exact causes are unclear, but it might be due the high concentration of arithmetic operations in Blowfish (CRC32 being dominated by Boolean ones) which performances suffer from several runtime checks (e.g. overflow).

However, when it comes to gather statistics, the simulation drastically slows down to 4 or 5 MIPS. In effect it is done through event handling which (1) adds a function call each time a target instruction is executed (2) uses the delegate mechanism, slower than classic method calls.

Results of the FFast/allb-b simulator with Blowfish are not shown: the high concentration of function calls makes sequences to miss very often; the system collapses due to multiple calls to the compiler which prevents statistic gathering (the situation gets better with FFFast/allb). Some tuning is still needed in this field, especially concerning the way sequences re-compilation decisions are taken.

V. CONCLUSION AND FURTHER WORK

This paper has presented FFast, an ISS written in C# which enables fast simulation speed through compiled ISS techniques. It illustrates the possibility of taking advantage of all the benefits which come with high level languages above VM in term of software engineering without impacting performances negatively too much. Those have been evaluated on the CLR only, but FFast runs also seamlessly on Mono. In effect, it can run at about 225 MIPS on a Core2 Duo at 2.6 GHz (6 with statistics gathering) which represents only a 10 time slowdown comparing to the hardware speed (500 with statistic gathering).

FFast uses C# then bytecode as intermediate representations between target code and n-code, and takes advantage of the ability of C# to quickly call its own compiler. FFast is also highly traceable at a very low engineering cost, thanks to the event paradigm provided by C#. The work presented in this paper introduced several techniques to reduce the overhead due to the presence of the VM. Further work would include emitting code in bytecode instead of C# during macro-expansion in order to have more control on the VM's behaviour. It would also include efficient support of more complex target architectures in C-ISS mode, as already supported by the interpretative

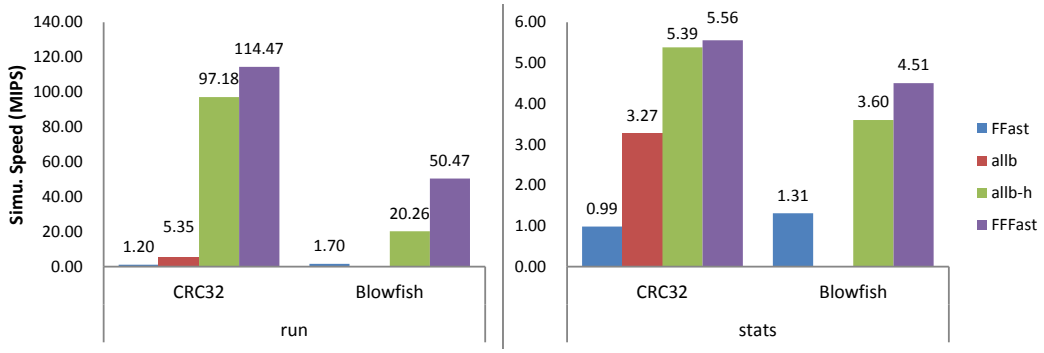

Fig. 2. Simulation Speed of FFast

counter-part of FFast. Another enhancement would be the support of the execution of operating systems.

Yet, the presence of the VM is indeed troublesome when it comes to evaluate performances as it adds some noise in measurements. Furthermore, simulation speed remains bellow equivalent simulators programmed using (especially) the C language and we think it will always be the case. In effect C# does not give close access to the host machine resources like C does, due to its high level programming model.

Still, this programming model offers a great opportunity for ISS developers to improve their efficiency, as well as for hardware designers to ease the design space exploration phase. Furthermore, the apparition of VM accelerated hardware (Jazelle ...) can potentially make this approach even more attractive.

REFERENCES

[1] A. Goldberg and D. Robinson, *Smalltalk-80: The Language and its Implementation*, M. A. Harrison, Ed. Addison-Wesley, 1983.

[2] R. B. Smith and D. Ungar, "Self: The power of simplicity," in *OOPSLA*, 1987, pp. 227–241.

[3] "Clr overview." [Online]. Available: http://msdn.microsoft.com/en-us/library/ddk909ch(VS.71).aspx

[4] "The mono project webpage." [Online]. Available: http://mono-project.com/

[5] J. Lapalme, E. Aboulhamid, G. Nicolescu, L. Charest, F. Boyer, J. David, and G. Bois, ".net framework: A solution for the next generation tools for system-level modeling and simulation," in *DATE*, vol. 1. IEEE Computer Society, 2004, p. 10732.

[6] "Simplescalar llc." [Online]. Available: http://www.simplescalar.com/

[7] L. Deutsch and A. Schiffman, "Efficient implementation of the smalltalk-80 system," in *Annual Symposium on Principles of Programming Languages*, 1984, pp. 297–302.

[8] E. Schnarr, M. Hill, and J. Larus, "Facile: a language and compiler for high-performance processor simulators," in *PLDI*, 2001, pp. 321–331.

[9] E. Schnarr and R. Larus, "Fast out-of-order processor simulation using memoization," in *PLDI*, 1998, pp. 283–294.

[10] V. Zivojnovic, S. Tjiang, and H. Meyr, "Compiled simulation of programmable dsp architectures," in *IEEE Workshop on VLSI Signal Processing*, 1995, pp. 187–196.

[11] J. Zhu and D. Gajski, "A retargetable, ultra-fast instruction set simulator," in *DATE*, 1999, pp. 363–373.

[12] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann, "A universal technique for fast and flexible instruction-set architecture simulation," in *DAC*, 2002, pp. 22–27.

[13] M. Reshadi, P. Mishra, and N. Dutt, "Instruction set compiled simulation: Technique for fast and flexible instruction set simulation," in *DAC*, 2003, pp. 758–763.

[14] ——, "Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation," in *ACM Transactions In Embedded Computing Systems*, vol. 8, no. 3, 2009, pp. 1–27.

[15] M. Burtscher and I. Ganusov, "Automatic synthesis of high-speed processor simulators," in *MICRO*, 2004, pp. 55–66.

[16] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.

[17] "What is nml ?" [Online]. Available: http://www.retarget.com/products/whatisnml.php

[18] "Events (c# programming guide)." [Online]. Available: http://msdn.microsoft.com/en-us/library/awbftdfh(VS.80).aspx