A Versatile Generator of Instruction Set Simulators and Disassemblers

Tahiry Ratsiambahotra, Hugues Cassé, Pascal Sainrat
IRIT – Université Paul Sabatier de Toulouse
Hipeac European Network of Excellence

Abstract— **Instruction-set simulators (ISS) are more and more used in design space exploration and functional software testing. Furthermore, cycle-accurate simulators are often made of a functional coupled to a timing simulator. Research about ISS generators is not new but most often addresses only simple instruction sets (i.e. RISC). This paper describes techniques to ease the description of complex Instruction-Set Architectures and to increase simulation speed. They are integrated in a tool which generates libraries containing functions to disassemble (useful for testing), decode and simulate many different architectures like RISC, CISC, VLIW and is able to deal with variable-length instructions. We successfully generated and used ARM/thumb, HCS 12X, Tricore, Sharc, PPC simulators and experiments have been made on the x86 architecture.**

Index Terms— **Instruction-set simulation, modeling language, instruction decoding, simulator generator.**

I. Introduction

Embedded system design is more and more based on simulators because of their early availability and ease of use. Testing people also would like to make tests on simulators as it allows making tests in parallel on a network of stations. Embedded systems are often composed of several different processors having different instruction sets. After the RISC era comes the extended RISC era where most of instruction sets are based on a RISC core but with many extra instructions dedicated to specific algorithms. In the mean time, CISC or DSP instruction sets still exist and 16-bit instructions become popular for energy savings.

Making an instruction-set simulator from scratch is quite long and error-prone. Thus, researchers have been looking for instruction-set description languages and simulator generators for quite a long time. Generating ISS for RISC processors from some description is quite easy. Several tools have been made in the past addressing different kinds of instruction sets but most are limited to a few kinds of instruction sets. Our description language, based on nML [1] and its associated tool allows us to address many different instruction sets.

This paper presents the various problems that are encountered in today's instruction sets and how we modify nML and our tool to deal with them and still generate efficient simulators and disassemblers.

The organization of the paper is the following. Section 2 presents related work whereas section 3 describes the intrinsic of the language and of our tool. Section 4 presents enhancements to the nML language that we feel useful either for disassembling correctly, for simulation performance and for representing various instruction sets. Before concluding, section 5 presents the simulation speeds according to these two features.

II. Related Work

The automatic generation of simulators starting from an architecture description language (ADL) drew the attention of many researchers since more than one decade. Most of these languages capture the complete architecture i.e. it models the instruction set and the hardware structure of the processor.

On the contrary, we think that the instruction set and the hardware must be described separatetely. Coupling them allows an easily implementation of a timing simulator.

The LISA language [1] and the associated commercial tool can generate several tools like a compiler, assembler, disassembler, debugger and linkage editor from a simple description. LISA is inspired by the nML language [2] which is also the base of our work. It is an instruction-centric language. Around the description of each instruction, several attributes are attached which captures the syntax assembler, the binary code, the behavior of the instruction and so on.

EXPRESSION [3] is an instruction-centric language which is based on a LISP formalism. For the structural part of the processor, EXPRESSION uses a graphical model which facilitates the description of the micro-architecture but is not very flexible. This tool can generate a simulator and a compiler.

LISA and Expression have been thought to generate new processors while our tool is more oriented towards software testing and simulation of existing architectures.

SIM-nML [4] is also a language based on nML. It has been developed at the university of Kanpur India. A SIM-nML model is similar to LISA as the functional units use is attached to each instruction by using the attribute "uses". It does not allow to model complex processor architectures.

All these languages use only one description to generate a complete simulator (architecture and micro-architecture).

If it is easy to describe simple processors, describing complex processors is almost forbidden with these tools. We believe that architecture (the programmer view which includes the instruction set and the memory model) and micro-architecture descriptions should be separated. The structural part can be described in a language more adapted like system C and can use functions provided by the ISS generator (fetch decode, execute).

The goal of Sleipnir [5] is similar. Sleipnir tries to tackle the problem of complex instructions found in the new architectures like ARM/thumb. Sleipnir can target variable-length instructions. To our knowledge, CISC architectures like x86 has not been tested with this language.

The generation of the decoder described in [6] is the closest to our technique. They use bitmasks to capture the opcode map of the architecture. But, we can't see how their model can deal with the existing opcode conflicts such as in an Arm/Thumb architecture.

Some papers deal with improving the simulation time of interpreted simulators.

We adopt the technique described in [7] to accelerate decoding by caching decoded instructions.

The second technique, called compiled simulation, is decoding offline which means transferring the decode time to the generation time of the simulator. It affords very good results but its use is restricted to simple instruction sets. Dynamic relocation or online possibility of changing endianness or code size like arm/thumb is very difficult if not impossible.

An improvement of this technique, called semi-compiled simulation, is described in [9]. The interesting idea is to add an option which will make it possible to decode after compilation if it is necessary. The drawback is that a simulator must be generated for each application.

Code translation [10] is one of the fastest techniques as regards simulation. Each instruction is represented directly into a host machine instruction but it might happen that some instructions are not translatable in the host instruction set. Nevertheless, this approach must always depend on interpretative simulation to disassemble or debug.

III. The Description Language and the Generator

A. The nML description language

nML is at the origin of several generators and languages like Lisa since it is well suited to describe instruction sets efficiently.

We also use nML but enhance it in several ways in order to generate efficient simulators for any kind of instruction set.

An nML description is made of three parts as shown in next textbox. The first part describes the state of the system (memory and registers). The second part describes the addressing modes in order to avoid describing them several times in the instructions which use them. The third part is the description of the instructions themselves. Instructions can be grouped into groups and part of their description can be described once.

Op introduces an instruction or a list of instructions connected by an OR (|) operator. An instruction has three attributes: syntax, image and action which corresponds respectively to the assembler syntax, the binary image and the code corresponding to the execution of the instruction

The nML grammar builds a tree to go through instructions starting from conjunctions (AND) and disjunctions (OR) relationships. In nML, OR is represented by | and AND by type or addressing mode encapsulation in the parameters.
A complete description of the nML language can be found in [10].

```
// State of the system
reg R[16,32] // 16 registers of 32 bits
mem [1024, 8] // 1 KB of memory

// Addressing modes
mode all_modes = register | imm
mode register(Index:card(4))
      syntax = format("R%d",index)
      image = format("%4b",index)
mode imm( rot: card(4), base:card(8)) = coerce(32, base <<< rot *2)
      syntax = format("#(%d, %d)", base, rot)
      image = format("%8b%4b",base,rot)

// Instructions
op ARM=  ALU | BRANCH | LOAD_STORE
op ALU = ADD | SUB | AND …
op ADD = ADD_i | ADD_reg
op ADD_i(dest: register, src:register, imm32:imm)
      syntax = format("ADDI %s,%s,%s", dest.syntax, src.syntax imm.syntax)
      image =
            format("11111010101%s%s%s",dest.image,src.image,imm.image)
      action = { dest = src + imm}
```

nML is very convenient but has some limits. Firstly, describing instructions with prefixes as in the x86 is tiresome and generates huge trees. Secondly, when there is a difference between the encoding of a parameter and the value mentioned in the assembler syntax, it is impossible to disassemble correctly. Most programmers have observed this problem in debuggers. Recognizing such an instruction is quite difficult.

The same problem appears when constants are coded in several non adjacent fields of bits as in the freescale HCS12X processor family. For example, in the decrement and branch instruction of the HCS12x, the sign bit of an immediate value is coded separately from the immediate value. Finding the immediate value from the binary code needs to apply a treatment to both fields.

IV. Enhancements

We experiment our tool with many instruction sets

(ARM, PPC, HCS12X, Tricore, Sharc and finally x86). During these experimentations, we collected all the features that were difficult or impossible to deal with or those which drastically increase the size of the simulator or the simulation speed.

From these observations, we decided to enhance the language and/or the tool in several ways which are described here.

A. The predecode attribute

This attribute might be used for several purposes: either to enhance simulation speed and decrease simulator size or to enhance the disassembler.

Most often, instruction decoding is done statically when generating the simulator. In our tool, instructions decoding is also static but part of it might be dynamic depending on the content of this attribute.

This attribute is also used to describe how to disassemble some instruction operands like an immediate value in the ARM instructions. The immediate operand is represented by 2 fields (base and rot) and an operation (base << rot * 2) has to be applied to these 2 fields to find the immediate value. The aim of such an encoding is to be able to code the most useful 32-bit immediate values in few bits.

In order to display the correct value when disassembling, the operation to apply has to be mentioned in the predecode attribute because this cannot be done statically or there would be one instruction for each possible operand value.

```
mode IMM (imm32:card(32),  rot: card(4), base:card(8))
      predecode = {imm32 = base <<< rot *2;}
      syntax = format("#%d", imm32)
      image = format("%8b%4b%0b",base,rot,imm32)
```

B. Parameters with dynamical syntax

In nML, the assembler syntax of a parameter should be statically defined. Conceptually, it sounds good but, faced to the reality of instruction sets, it is preferable that a parameter has its own syntax and, furthermore, that it might be dynamically defined during decoding. Finally, it is preferable that this syntax may be conditional. For example, one can describe the syntax of a register parameter in an IA32 instruction as:

```
predecode = {
      if prefix == « 0x66 » then
            register.syntax = 'AX'; else
            register .syntax = 'EAX';
      endif ;  }
```

C. Pseudo parameters

nML supposes a strict correspondence between the image and the syntax. This is not always the case as in the immediate field of the ARM which is described by two fields in the binary code (base and rot as mentioned before) and one parameter in the assembler syntax (the value resulting of base << rot * 2). Thus, in our tool, a parameter may have no image (declared as %0b) and this parameter can be used as a variable to keep the result of a predecode calculus like base << rot*2. Thus, the correct value is shown when disassembling.

D. Dynamic endianness

Many processors today can use both little and big endian and, more, can switch dynamically between two modes by modifying the endian bit in a special register.

Our tool has some special files that can be used to implement various C functions such as the adaptation between system calls in the benchmark and the system calls of the host (system functions are not simulated but emulated by the system of the host) and many other features like virtual memory.

Thus, it is possible to specify the function machine_is_little in such a file and mention in the nML description that memory accesses should check the endianness.

```
mem MEM_DATA[ ;;;]
MEM_DATA_is_little = machine _is_little()
```

E. Several decoders in the same simulator

The main function in a functional simulator is instruction decoding. That's where most time is spent and it's the most complex function. A good tool is supposed to support any instruction set and it helps in debugging the description by detecting conflicts i.e. several instructions having the same opcode.

1) Basic algorithm

Let us have a look at some instructions of the HCS12X to illustrate the decoding algorithm.

syntax	Binary image	Size (bits)
tsta	10010111	8
adca	1001100100000000	16
ldd #n	11001100xxxxxxxxxxxxxxxx	24
tbeq SP, #rel9	0000010001000111xxxxxxxx	24
ibeq A, #rel9	0000010001000000xxxxxxxx	24

x bits are bits corresponding to the parameters. Their value will be defined during decoding and not statically.

A mask is built for each instruction with 1s for the known bits and 0s for the unknown bits.

syntax	mask
tsta	11111111
adca	1111111111111111
ldd #n	111111110000000000000000
tbeq SP, #rel9	111111111111111100000000
ibeq A, #rel9	111111111111111100000000

To build the decode tree, we first define the global mask by anding the masks of all the instructions. The global mask should not be equal to 0 otherwise there's an opcode conflict.

Here, the global mask is: 111111110000000000000000

Each possible value of this mask (here 255 as the mask contains eight 1s) corresponds to an instruction. To each instruction, we can associate a local mask by eliminating the global mask in the mask of the instruction.

For example, tbeq and ibeq corresponds to the value 4 (global mask and binary image) and their local masks are: xxxxxxxx11111111xxxxxxxx

Anding this local mask with the binary images gives 0x47 for tbeq and 0x40 for ibeq.

This process is repeated until the value obtained corresponds to only one instruction. We thus obtain a decoding tree as shown in Figure 1.

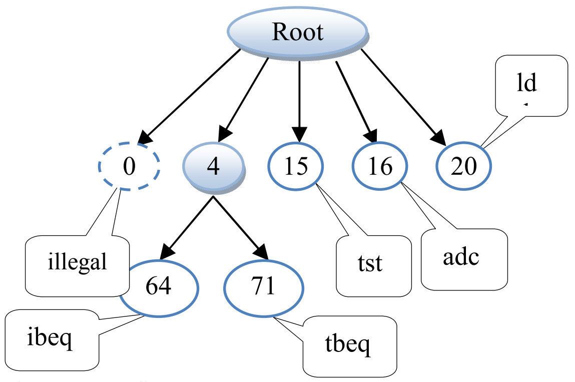

Figure 1. Decoding tree

2) Selectable ISS

In some processors, several instructions might have the same opcode as the opcode is completed by an external information. For example, in the ARMV5 architecture, the T bit in the CPSR register points out the fact that the fetched instruction is 16 bits or 32 bits.

We added a version attribute to the instructions. Then, our tool generates several decoders if there are several version values. The variable in the nML description which defines which decoder should be used is defined in the command line of the tool.

Considering the ARM processor, we have:

```
reg Ucpsr [1, card(32)]
reg T_Bit[1, card(1)] alias Ucpsr[5]

op arm_ver(ar :ARM)
      syntax = ar.syntax ;
      image = ar.image
      action = ar.action
      version = « 0 »
op ARM = ALU_arm | Branch_arm
op thumb_ver (th :THUMB)
      syntax = th.syntax
      image = th.image
      action = th.action
      version = « 1 »
op THUMB _ ALU_thumb | Branch_thumb
```

This can be easily extended to more than 2 instruction sets, for example a processor with several coprocessors or configurable processors.

F. Examples of use of the predecode attribute

This paragraph illustrates the use of the predecode attribute in order to disassemble correctly and factorize the description.

1) ARM architecture

We illustrate the use of the predecode attribute through one of the addressing modes of the ARM architecture.

In the ARM architecture it is possible to write an instruction such as:

ADD R5,R6,R7, ROR #2

In this instruction, R7 content is rotated right by 2 before being added to R6, the result being stored in R5. However, ROR #0 is in fact RRX.

To disassemble correctly such an instruction, the syntax should be different according to the immediate value (ShifAmt) and the kind of shift (ShiftKind). This can't be described in nML but is possible with the predecode attribute as shown below. Thus, the predecode attribute enhances disassembling and reduces the number of nodes.

```
mode REG_INDEX ( r : index ) = r
      syntax = format( "r%d", r )
      image  = format( "%4b", r )

mode instructionSpecifiedShift(rm:REG_INDEX,shiftAmt:Bit5,shiftKind: Bit2)
      predecode = {
            switch (shiftKind) {
                  case 0 :   if (shiftAmt != 0 ) then
                                    shiftKind = 'LSL';
                              else
                                    shiftKind = '';
                              endif;
                  case 1: shiftKind = 'LSR';
                  case 2: shiftKind = 'ASR';
                  case 3:   if (shiftAmt != 0 ) then
                                    shiftKind = 'ROR';
                              else
                                    shiftKind = 'RRX';
                              endif;
                  };
            rm.predecode; }
      syntax = format("%s, %s #%d", rm, shiftKind, shiftAmt)
      image  = format("%5b%2b0%s", shiftAmt, shiftKind,rm.image)
```

2) Illustration on the IA32

The IA32 is probably the most complex instruction set as it has evolved for dozens of years and is still compatible with earlier versions.

We illustrate here how we can describe such an instruction set in our enhanced nML.

The main problem again is that the addressing mode is made of several non adjacent fields, all of them being optional and their behavior also depending on the presence of prefixes before the opcode.

a) Direction bit

X86 instructions have two operands. In most two-operands instruction sets, the destination is the first operand. In the x86, it depends on an additional bit called d.

In nML, we would have to describe two ops, one for each direction but these two operations would be almost similar differing only in the order of the operands.

In gliss, the generator we developed, we can describe such instructions with one op by using the concept of parameters with dynamic syntax as shown in the example below. The size of the tree is then half the size of the original one.

```
op ADD(d : card(1), reg_1st : reg32, reg_2nd :reg32, dest :card(1), src :card(1))
    predecode = {
        reg_1st.predecode;
        reg_2nd.predecode;
        if d == 1 then
            dest.syntax = reg_1st.syntax;
            src .syntax = reg_2nd.syntax;
        else
            dest.syntax = reg_2nd.syntax;
            src .syntax = reg_1st.syntax;
        endif ; }
    syntax = format(« ADD %s, %s », dest.syntax, src.syntax)
    image  = format(« 000000%1b111%s%s%0b%0b », d, reg_1st.image,
reg_2nd.image, dest,src)
    action = {
        if d == 1 then
            reg_1st = reg_1st + reg_2nd;
            update_flag(reg_1st);
        else
            reg_2nd = reg_2nd + reg_1st;
            update_flag(reg_2nd);
        endif; }
```

In this example, update_flag is a macro described elsewhere in the nML description.

b) Dealing with prefixes

IA32 instructions may have prefixes which are before the opcode. Among other things, they determine the way addressing mode fields should be interpreted or specify the action of the instruction.

An instruction may have no prefix or several prefixes.

With the pseudo-parameters and the dynamical syntax, we can deal with these prefixes at a reasonable cost and ensure a correct disassembling. In pure nML, it would produce an enormous tree (several hundreds of MBs).

We need to use a global variable (prefix_flag) declared as a reg. This variable will recall which prefixes the instruction has.

```
reg prefix_flags [1, card(8)]
```

Special addressing modes will update these flags and the addressing mode will read these flags.

For convenience, prefixes are described in groups. Here we describe only group one.

```
// group one is made of four prefixes which are stored in the two upper bits of
prefix_flags

mode no_one(n:card(1))
    predecode = { prefix_flags<7..6> = 0 ; }
    syntax = ""
    image = format("%0b",n)
mode lock()
    predecode = { prefix_flags<7..6> = 1;}
    syntax = ""
    image = "11110000"  // 0xF0
// here is the decscription of repne (not developped here)
mode repe()
   predecode = { prefix_flags<7..6> = 3; }
    syntax = ""
    image = "11110011"  //0xF3

mode group_1 = lock | repne | rep | no_one
mode  opsize()
    predecode = { prefix_flags<1..1> = 1; }
    syntax = ""
    image = "01100110"  //0x66
```

There are three other groups which are described the same way.

As prefixes of different groups might affect an instruction, the prefix mode calls the predecoding of each group.

```
mode prefix(pref1:group_1,pref2:group_2,pref3:group_3,pref4:group_4)
    predecode = {
        pref1.predecode;
        pref2.predecode;
        pref3.predecode;
        pref4.predecode; }
    syntax = ""
    image=format("%s%s%s%s",pref1.image,pref2.image,pref3.image,pref4.imag
e)
```

The description of the real addressing mode reads prefix_flags and behaves according to its content.

```
mode RM_Addr_Simple(RM:card(3),regi:card(3))
    predecode= {
        if prefix_flags<1..1> == 0 then
            switch (regi){
                case 0:
                    regil = '[EAX]';
                ....
            };
        else
            switch (regi){
                case 0:
                    regi = '[AX]';
                ......
            };
    }
    syntax = format("%s,%s",regi.syntax,RM.syntax)
    image = format("00%s%s",regi.image,RM.image)
```

```
mode mode_RM = RM_Addr_Simple | ...
```

Then, each IA32 instruction can be described as follows.

```
op ADD (pref :prefix, d :card(1), reg1: reg32,  modrm: mode_RM, ... , depl:
displacement, imm: immediate)
      predecode = {
            pref.predecode ;
            modrm.predecode ;
            sib.predecode ;
            .....
      }
      syntax= format(« ADD %s ,%s »,reg1, modrm.syntax, depl.syntax,
imm.syntax)
      image = format (« %s000000%1b1%s%s%s », pref.image, d,modrm.image,
sib.image, depl.image)
      action = { reg1 = reg1 + M[modrm + depl] + imm ; }
```

V. Results

System calls are emulated by the systems calls of the host machine. As we did not implement all of them, we choose the five following benchmarks: qsort, basicmath and bitcnts from the mibench and adpcm and li from the spec95.

All measures have been realized on a DualCore@3GHz but only one core has been used as the simulator is sequential. We realized these measures for an ARMv5 nML description. The benchmarks were compiled with gcc.

A. *Fetch buffer and Decode cache*

Accessing the memory of the simulator is quite complex for various reasons due to the automatic generation. To decrease access times to the memory, we have added a fetch buffer and several instructions are fetched from the memory at once. Once in the fetch buffer, accessing an instruction is very simple and efficient.

As shown in

Table 1, adpcm has a special behavior as it is made of a small loop which is contained in the buffer. Thus, above 20, the performance is the same. Otherwise, each benchmark has its own best size but the results show that a size of 5 to 10 gives most of the benefit and the increase is quite small above this size.

Another well-known technique for enhancing the performance is the decode cache: the last decoded instructions are kept in this software cache in their decoded form. It avoids going through the decoding tree to find an instruction. Our results, in Figure 3, confirm previous results [11].

Buffer Size	*1*	*5*	*10*	*20*	*30*	*40*	*50*
qsort_small	5.5	7.5	7.9	7.9	8.1	8.1	8
qsort_large	6	8.5	8.7	8.9	9	9.1	8.9
basicmath_small	6	8.7	9.1	9.3	9.5	9.4	9.3
basicmath_large	6	8.6	8.9	9.1	9.2	9.2	9.1
bitcnts_small	6.5	9.7	10.2	10.3	10.6	10.6	10.4
bitcnts_large	6.5	10	10.3	10.6	10.7	10.7	10.6
Adpcm	4.4	6.1	9.9	9.9	9.9	9.9	9.9
Li	5.3	7.3	7.5	7.6	7.7	7.7	7.6
Average	5.9	8.5	9.3	9.5	9.6	9.6	9.5

Table 1. MIPS Performance according to the fetch buffer size

	Flat nML	**nML with predecode**
Decode leaves #	8,453	300
Generation time	14’	55”
Library source code size	154 MB	3.4 MB
Library binary size	44 MB	0.72 MB

Table 2. Characteristics of the library

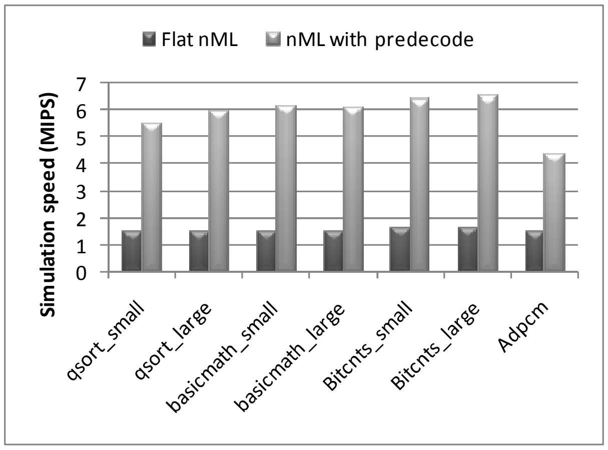

Figure 2. Simulation speed of an ARM simulator without and with predecode

B. *Library characteristics*

Table 2 shows the characteristics of the generated library for a flat nML description and for a nML description with the predecode statements.

Not surprisingly, the size of the library is much smaller with the predecode. As the number of leaves is much smaller, a lot of code is not replicated. Moreover, as shown in Figure 2 using correctly the predecode attribute provides a fourfold increase of the simulation speed. The extra code executed at decoding is much less costly than going through the very deep tree.

C. *Simulation speed*

Figure 3 shows the simulation speed for the basic generator, the generator with the fetch buffer, the decode cache and finally with the decode cache and the fetch buffer.

The fetch buffer alone increases simulation speed but not much because loss of performance is dominated by decoding. The decode cache considerably enhance simulation speed and using the fetch buffer is more profitable. With both features, simulation speed is enhanced by 4 to 5 according to the benchmarks.

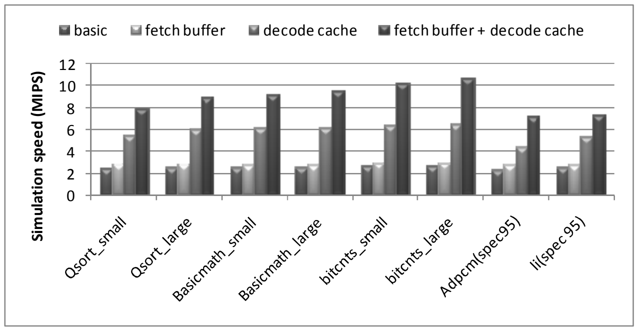

Figure 3. Simulation speed

Figure 4 compares gliss to two existing "hand-made" simulators: simplesim, the ARM functional simulator of simplescalar 4.0 which is very much used in the architecture research community and gdb-armulator 6.7.

Although our simulators are generated automatically and thus can be produced much more quickly than to completely write, their speed is higher than for interpreted simulators which have been written from scratch. Note that simplesim is not able to simulate basicmath_small.

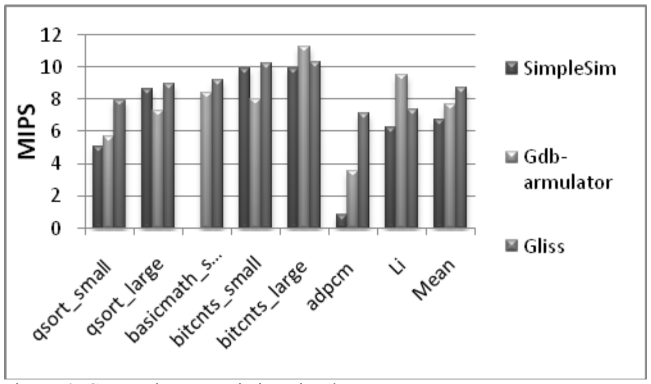

Figure 4. Comparison to existing simulators

VI. Conclusion

Our tool generates a library of functions. These functions can be used with a very simple simulator consisting in a dozen of lines to do functional simulation. They can also be used in a performance-oriented simulator where timing is modeled (written in systemC [12] for example). The timing part may deal only with timing, leaving functional aspects to the library by calling correctly its functions or it may also cover the functional part with its own state (registers and memory) for speculative processors or to check its state against the functional one.

Today, only the basic tool is distributed as the enhanced tool still needs tests to ensure its robustness. Furthermore, we would like to further increase the simulation speed before the release. We expect to distribute it in a few months with the descriptions of the six instruction sets we have partly or completely described.

Our tool produces an interpretive simulator which is not as fast as compiled simulation but has the advantage of its versatility, ease of description of a new instruction set and possible use for various tools such as a disassemble or for static analysis.

VII. References

[1] Freericks, M., *The nML Machine Description Formalism.* Fachbereich Informatik. Berlin : TU, 1991. Technical report. 15.

[2] A. Hoffman, T. Kogel, A. Nohl, G. Braun, O. Scjliebusch, O. Wahlen, A. Wieferink, and H. Meyr., "A novel methodology for the design of Application-Specific Instruction-Set Processors(ASIPSs) using a machine description language." Transanctions on Computer-Aided Design of Integrated Circuits and Systems, s.l. : IEEE, November 2001, Issue 11, Vol. 20, pp. 1338-1354.

[3] Fauth, A, Van Praet, J et Freericks, M., "Describing Instruction Set Processors using nML." Paris : IEEE, 1995. European Design and Test Conference. pp. 503-507.

[4] Prahbat Mishra, Frederic Rousseau, Nikil Dutt, Alex Nicolau., "Architecture Description language Driven Space exploration in the presence of Coprocessor." 2001. Tenth Workshop on Synthesis And System Integration of MIxed Technologies (SASIMI).
http://www.ics.uci.edu/~copper/publications/sasimi2001.pdf.

[5] V. Rajesh, Rajat Moona., "Processor modeling for Hardware Software Codesign." Goa : IEEE, 1999. 12th International Conference on VLSI Design. pp. 132-137. http://citeseer.ist.psu.edu/rajesh00processor.html. 0-7695-0013-7.

[6] Jeremiassen, Tor E., "Sleipnir- An Instruction-Level Simulator Generator." Austin, TX : IEEE, 2000. International Conference on Computer Design. pp. 23-31. http://ieeexplore.ieee.org/iel5/7044/18963/00878265.pdf?arnumber=878265.

[7] Mehrdad Reshadi, Nikhil Bansal, Prahbat Mishra, Nikil Dutt., "An efficient Retargetable Framework for instruction-set simulation." Newport Beach, CA : s.n., 2003. 1st International Conference on Hardware/Software Codesign and System Synthesis. pp. 13-18. http://portal.acm.org/citation.cfm?id=944645.944649. 1-58113-742-7.

[8] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, A. Hoffman., "A Universal technique for Fast and

Flexible Instruction-Set architecture simulation." New Orleans : IEEE Computer Society, 2002. 39th Design Automation Conference (DAC'02). p. 22. http://doi.ieeecomputersociety.org/10.1109/DAC.2002.1012588. 1-58113-461-4.

[9] Mehrdad Reshadi, prahbat Mishra, Nikil Dutt., "Instruction-set Compiled simulation: a technique for fast and flexible instruction set simulation." DAC, 2003. http://www.cecs.uci.edu/conference_proceedings/dac_2003/reshadi_instruction.pdf .

[10] Gulur, Nagendra., Novel Techniques for Very High spees instruction-set simulation. *Design Reuse.* [En ligne] www.us.design-reuse.com/articles/13718/. http://www.us.design-reuse.com/articles/13718/novel-techniques-for-very-high-speed-instruction-set-simulation.html.

[11] Clarke, B., Czezowski, A. & Strazdins, P., "Implementation aspects of a SPARC V9 complete machine simulator." Melbourne : Australian Computer Society, 2002. `Computer Science 2002', Vol. 4 of Conferences in Research and Practice in Information Technology. pp. 23-32.

[12] , *http://www.systemc.org/.*