FAST INSTRUCTION-ACCURATE SIMULATION WITH SIMNML[1]

H. Cassé, J. Barre, R. Vaillant and P. Sainrat

{casse, barre, vaillant, sainrat} @irit.fr
Université de Toulouse, Institut de Recherche en Informatique de Toulouse (IRIT)

Instruction Level Simulation has received big attention as it allows out-of-silicium test and hardware exploration. In this paper, we present GLISS2, the second release of a simulator generator based on the NML ADL. Thanks to the implementation of a set of optimization techniques (acceleration in memory emulation, caches for the decode step and blocking of instruction descriptors), we multiply by an average factor of 10 the simulation performances. Additionally, although the experimentation has only been made for the PowerPC, the performed optimization extends naturally to any instruction set described in NML.

***Index Terms*—Computer Science, Microarchitecture, Simulation**

I. INTRODUCTION

Instruction level simulation has received a big attention in the last ten years as it provides an easy and cheap way to analyze the behavior of computer systems. These simulators have been used in domains as different as architecture exploration (to estimate performances, power, etc), embedded systems validation (test without the actual hardware) or program emulation on a host system with a different architecture.

There are different ways to implement Instruction Set Simulators (ISS). Initially, such simulators were implemented by hand but their development was usually long and painful. More modern approaches use the description of the Instruction Set Architecture (ISA) in an Architecture Description Language (ADL) in order to generate the simulator automatically. Although the latter approach improves the productivity and makes the debugging of the obtained simulator easier, the produced simulators were often much slower than hand-written ones.

In this article, we want to show that the performance gap between automatically generated and hand-written simulators can be reduced if a special attention is payed to the way the code is generated and the different simulation steps are implemented.

The first section shows the generation process of our ISS generator, called GLISS2, second generation of GLISS [1] and the simulation process. Section III presents the different optimizations performed on the memory module, on decoding, and on the execution step. After the presentation of the related works, we conclude in the last section.

II. GENERATION AND SIMULATION

This section presents the ADL used in GLISS2 and the basics of the simulator generation.

A. SimNML

NML is an ADL, firstly described in [2], that allows to describe the ISA of a microprocessor in a synthetic and smart way, alleviating this painful and error-prone task. It provides also a specific syntax to model the instruction execution at the micro-architecture level but this extension is rarely used in practice as it is not adapted to the complexity of the current architectures. Yet, NML has been largely successful in implementing a lot of ISA as different as RISC (PowerPC, ARM, Sparc, TriCore) or CISC (M68HCS12, x86) or to derive machine code utilities as simulators, disassemblers or compiler back-ends [13]. NML is also used to implement instruction decoding in machine-level static analyzer like OTAWA [14].

NML describes the ISA as a collection of types, state items, modes and operations. Type system is bit-accurate and allows to qualify single registers, register banks and memories. State item aliases allow also to mimic some register configurations (like x86 registers) or to maintain type safety when accessing memories with data of different types.

Modes and operations provide a synthetic and concise way to describe the machine instructions. They are organized in a so-called AND-OR tree. An AND node is a node defining an instruction or a family of instructions from a set of parameters and attributes that are shared by all the derived instructions. This allows factorization of some description parts. OR nodes are used to describe alternatives, for example, in a family of instructions. Modes are used to describe state accesses like addressing modes while operations describe the actual instructions. To better understand the AND-OR tree structure, the example in listing 1 is an excerpt from the PowerPC description showing the initial node of the tree. It takes as parameter the set of all instructions but factorizes the common behavior of PC incrementation.

```
op instruction (x: allinstr)
        syntax = x.syntax
        image = x.image
        action = {
                NIA = NIA + 4;
                x.action;
        }
op allinstr  = uisa_instr
        |       vea_instr
        |       oea_instr
```

LISTING 1 SimNML Sample

[1]The works described in this paper have been developed in the context of the SOCKET FUI Project.

This sample shows also the standard attributes including disassembly syntax, binary image and semantics action of the instructions. The latter attribute, describing the execution behavior, is expressed in an algorithmic-like language for statements and in a C-like syntax for expressions (for bit-level computation). This leads to a language relatively close to the dialects used in the ISA handbooks.

B. Generation

The generation is performed by GLISS2 on the NML ISA description to produce C sources. The usage of the C programming language ensures good portability and performances of the simulator because (1) the produced code is very close to the host hardware and allows finer control on the computation and (2) it benefits from good optimizing compilers. The sources are then compiled to make a library or to build stand-alone simulator or disassembler. The library may be embedded in any program requiring any service supplied by GLISS2 (simulation, disassembling, instruction decoding). In addition, C generation strengthens the portability of the obtained library as a lot of running environments use it as an interface.

To improve performances, the key word of the generation process was: remove all dynamic computations if they can be computed at compilation time. Indeed, it is incredibly easy to obtain code that was accounted to be optimized out by the compiler but that is not because of compilation bounds like compilation units, library organization or memory accesses.

One good example is the management of the endianness if it is different on the simulated and on the host architectures. A naive implementation of this function, shown in Listing 2 tests it at run-time:

```
int is_host_little(void) {
        union { uint8_t c[2];
                uint16_t w; } u;
        u.w = 0xffaa;
        return u.c == 0xaa;
}
uint16_t w = read_word_from_memory(address);
if(is_host_little())
        invert_word(w);
```

LISTING 2 Naive Endianness Test

The idea supporting this piece of code is that (1) different specialized versions of sources are not needed and (2) such a small function will be automatically in-lined by the compiler each time it is called. The problem is that most C compilers are not able to perform such an optimization if the function is located in a different compilation unit. One may also observe that, even if the function is in-lined, this approach is a waste of time: simulated and host endianness is known at compilation time and the exact behavior of the simulator code can be determined at this moment. This is easily achieved using C preprocessor directives because GLISS2 is able to obtain the host endianness at generation time to configure the compilation behavior accordingly, as shown in Listing 3.

```
uint16_t w = read_word_from_memory(address);
#if HOST_ENDIANNESS != TARGET_ENDIANNESS
        invert(w);
#endif
```

LISTING 3 Endianness with Preprocessor

The performance gain is small at unit level but such an operation is usually repeated tenth of thousands times in a simple simulation and may finally cause important waste of time.

In the same way, we specialize the generated code as much as possible. SimNML provides a versatile way to handle bit fields using the syntax: *item<upper bound . . lower bound>*. Such a syntax allows to handle bit fields in expressions or to modify them when put in the left hand side of an assignment. In addition, this operation supports dynamic determination of bounds to get as complex operations as bit field inversion if the upper bound is less than the lower bound.

Although an all-in-one function is required to implement the more complex forms of this operation, most calls of this operator exhibits simpler behaviors: bounds are constants or give access only to one bit. In these cases, costly calls to a complex function are replaced by specialized in-lined expressions with pre-computed masks. This basic optimization has shown to make an average performance gain of several MIPS (Million Instruction Per Second).

This philosophy is applied all over the different modules composing the simulator. For example, fixed-size instruction set decoding, as found in RISC, is far more simple to handle than variable-size CISC instruction sets and, therefore, uses a simpler implementation of the instruction decoder.

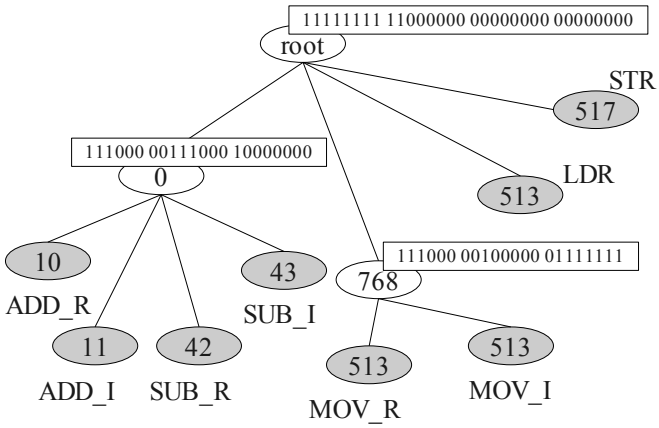

Figure 1: Decoding Tree

C. Simulation

The realization of the instruction simulation is performed in three steps: (a) fetch, (b) decoding and (c) execution.

The fetch step consists in accessing the program counter of the program, to get the instruction word from the memory and to match it with an instruction described in the ADL. If no match can be established, the instruction is considered as undefined and an error may be raised according to the running environment.

The complexity for fetching instruction comes from the

number of instructions usually found in an ISA and from the fact that a bit, in an instruction word, may be used either to encode the operation code, or an argument in order to keep the instruction encoding compact. Usually, the instruction are grouped in a hierarchical tree of instruction sets that share the same value for a specific set of bits. The leafs of this tree identify specific instructions. GLISS2 automatically extracts this tree from the instruction images and generates an optimal tree (in terms of height) as shown in Figure 1.

Each non-leaf white node of the tree contains a mask (in white boxes) identifying involved bits and an array giving the children nodes according to the value of these bits. Leaf gray nodes only give the code of the recognized instruction. The traversal of this tree is pretty fast with fixed-length ISA but more complex with variable-size ISA as the length of the instruction depends on its operation code. Therefore, the tree traversal may require to load additional bytes from the memory what slows down the fetching process. One may also notice that the fetch speed depends also on the complexity of the used masks that may require several shifts and mask operations to get index on the children nodes.

At this point, the instruction has been identified and specific processing can be applied. The decoding phase consists in calling an instruction-specific function that extracts the arguments of the instruction to provide them to the execution phase. It is implemented as bit field accesses to the arguments encoded in the instruction word. The efficiency of this phase depends on the number of arguments to retrieve as it is implemented as a set of masking and shifting operations.

The final step performs the actual execution of the instruction. The function implementing the instruction is called and uses the decoder arguments of the previous phase. Usually, the action attribute is translated in C code whose main effect is to modify the state of the simulation possibly including memory accesses for data.

Three main components are involved in the instruction execution and impact the simulation time. The instruction execution function is optimized during the generation phase but we will see that it remains place for improvement. The memory is either used in the fetch step, or during the execution to read or write data in memory: good performances will have an effect over all the simulation process. For the decoding step, one may observe that, usually, the instructions do not change during the simulation while, in our process, each instruction is decoded each time it is executed. Consequently, factorizing all or part of these extraneous decoding should produce gain.

III. Optimizations

This section presents the efforts performed on GLISS to improve performances. The different strategies have been evaluated with 4 sets of benchmarks:

1. integer benchmarks of Mälardalen [3] (MINT),
2. float benchmarks of Mälardalen [3] (MFLT),
3. the automotive set of MiBench [4] (AUTO),
4. a selection of SpecInt95 benchmarks [5] (gcc, crafty, mcf, parser).

The selection in the different series of benchmarks has been mainly constrained by our implementation of system calls. The host machine was an Intel Core 2 Duo at 3 Ghz with 4 Gb memory running Linux Mandriva 2010.

A. The Memory Module

The memory is one of the most used module during the simulation. It is used at fetch time to get instruction words or during the execution step when the instruction performs memory accesses. Therefore, its implementation has a straight impact on the simulation performances.

Other challenges of the memory implementation include support of endianness and of the address space. For the former, a well-known technique is used: byte order is inverted in the memory page so that scalar data reading is the same operation in the simulated language and on the host machine at the small cost of fixing the offset in the page[2].

Support of simulated address space of equal or bigger size than the one of the host machine is also an important issue. The first statement is that the memory cannot be implemented as a simple array of bytes.

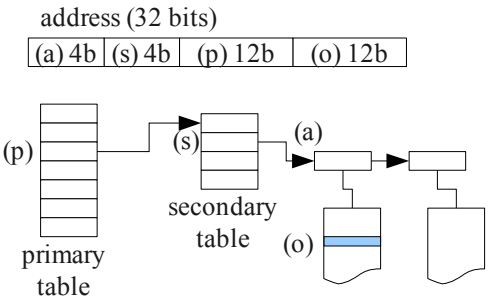

Figure 2: Memory Access

In GLISS2, the memory is structured in pages and only the used pages are represented. To retrieve the page matching an accessed address, two indirect arrays are used as represented in Figure 2. The address is split in different areas whose size may be configured at compilation time (as default we are using 4Kb pages, 4096 entries primary table and 16-entries secondary tables). The (p) part indexes the primary table and allows to obtain the secondary table. This one, indexed by the (s) part, gives a linked list of pages matching the different memory areas (a). Once the matching page is found, the accessed word is read from the page using the offset part (o).

The part sizes proposed in Figure 2 give a good tradeoff between access speed and memory usage. Yet, as experimental results have shown that a lot of time is spent in the tables access, we also implemented a memory module release with only one level of indirection: as a side effect, the memory footprint is larger with a table of 256 Kb on 32-bits machines (instead of 16 Kb in the two-level implementation).

[2] Notice that GLISS2 only supports little and big endian encoding: other byte ordering are currently mostly no more used.

Table 1: Memory Implementation Comparison

Benchmark	2-level (Mips)	1-level (Mips)	Gain (%)
MINT	5.89	6.49	10.13
MFLT	5.61	6.12	9.09
AUTO	5.55	6.01	8.37
crafty	4.93	5.17	4.87
gcc	6.02	6.77	16.28
gzip	6.17	6.83	10.6
mcf	5.99	6.64	10.85
parser	5.88	6.26	6.46
average	5.76	6.29	9.58

The Table 1 compares the performances of the two proposed memory implementations, 2-levels or 1-level. With an average gain of 9,58%, the 1-level memory implementation shows better results and will be used in the following.

B. Instruction Decoding

Instruction decoding is another costly step during simulation. Its complexity depends on the number and the encoding of the arguments and on the size of the instructions. It is basically implemented as a set of shifts, AND and OR operations to extract the bit fields from the instruction word and to recompose the instruction arguments. To fit well with the C language working, sign extension may also be required. The resulting arguments are then stored in a data structure, the instruction descriptor, that is used during the execution step.

A straight approach to reduce cost of decoding is to avoid repeated decoding for the same instruction. In fact, such a case is very frequent because most execution time is spent in loops where the same instructions are executed ever and ever. To benefit from this property, we have extended our decoder with a cache storing the already-decoded instruction descriptors. When the instruction is in the cache, the descriptor is directly passed to the execution step. In the opposite, the usual costly work is performed by fetching and decoding the instruction and the obtained instruction descriptor is stored in the cache.

We have experimented two types of keys for indexing the cache. The first one is the address of the instruction: it is straight-forwardly got from the PC register and quickly handled for accessing the cache. With the second key, we wanted to get benefit from using the same descriptor for instructions at different addresses but with the same encoding bytes, that is, the same instruction with the same arguments. The gain would include the decoding time but also a better use of the host data cache. It is achieved by the use of the instruction word as the table index what is relatively easy with a RISC processor but more complex with variable-size ISA. Whatever the instruction size, the real bottleneck comes from the memory access required each time an instruction is executed that is so costly that it cancels any benefit from the descriptor factorization.

So, we only retained cache indexing by address and have experimented different ways to realize the cache. For all implementations, the cache is structured in a set of power-of-two sets, that allows quick extraction of the index from the address. Each line is composed as a linked list of instruction descriptors for faster re-organization of the list.

Table 2: Decoding with Cache

Benchmark	Inf (× 100%)	FIFO (× 100%)	LRU (× 100%)
MINT	8.57	7.19	9.06
MFLT	5.75	5.42	6.03
AUTO	5.23	4.62	5.58
crafty	10.2	9.44	11.47
gcc		5.55	7.31
gzip	6.26	5.22	6.95
mcf	6.68	5.92	7.27
parser	5.33	4.42	5.78
average	**6.86**	**5.97**	**7.43**

The first implementation, INF, considers an infinite cache that may be viewed as the top of obtainable performances although this is not ever true because of effects depending on the host machine data cache. Other implementations exhibit a finite cache with different replacement policies: FIFO and LRU (Least Recently Used). Results of this comparison are shown in the Table 2 that displays speed multiplier factor relative to the performances without any decode cache and 1-level memory. LRU is the best method with an average factor of 7,43 (743 %) of gain. One may also observe (1) we have no result with an infinite cache for GCC due to memory exhaustion and (2) that even the worst cache results provide an improvement of 4,42: the caches are essential in performance look-up.

Even if the caches improve greatly performances of the decoding step, the retrieval of an instruction descriptor remains costly as it may include the traversal of a linked list. Yet, one may observe a unity of execution between instructions. This property is well known in optimization compilers where the program is represented as Control Flow Graph (CFG). To reduce the size of the CFG, there is not a vertex for each instruction but for a group, usually called Basic Block (BB). A BB is a sequence of instructions such that all instructions are executed as soon as the first one is executed and, therefore, only the last instruction can be a branch. From a simulation point of view, this means that we can generate blocks of instruction descriptors matching instruction block and we have to call only once the decoder routine to get it and execute the block instructions.

We have tested two implementations of this concept. In the first one (STRACE), we have considered blocks of fixed size, possibly crossing BB bounds, but easier and faster to process. In DTRACE, we are completely implementing the concept of BB but at the price of (1) variable-size blocks and (2) the requirement to know which instructions are branches to bound blocks. Fortunately, this is easily implemented in NML by adding a specific boolean attribute. The table 3 compares the different approaches giving the speedup improvement in percent compared to the LRU only cache performances. On the average, DTRACE seems to do a bit better but, in some cases as gcc or crafty, STRACE takes largely over.

Although we may have expected more performances from a

Table 3: Decoding with Block Cache

Benchmark	STRACE (%)	DTRACE (%)
MINT	26.36	39.25
MFLT	17.8	32.96
AUTO	46.9	52.24
crafty	14.36	2.26
gcc	35.59	24.64
gzip	32.23	41.79
mcf	24.01	27.24
parser	48.06	52.32
average	**30.66**	**34.09**

simpler implementation as STRACE, the cost of calling more often the decoding routines, on x86, seems to be unable to balance the more complex processing of the DTRACE. This statement is even more exploited in the following.

C. Instruction Execution

Instruction execution step can also be optimized. The first optimization comes from the C compilation: any optimization can benefit to the generated simulator. Secondly, we can also act on the way an instruction is executed: basically, execution of instructions involves two features on the host machine microprocessor: instruction cache and branch predictor.

The instruction cache provides fast access to the host machine instructions used in the simulation functions implementing a particular instruction. This mechanism is automatic and provides usually good performances thanks to the temporal and spatial locality of programs. In case of a simulator, this work may cause performance loss if two functions implementing very frequent instructions are in conflict as they are mapped to the same cache sets.

A solution would be (1) to obtain a profile of instruction frequencies and (2) to avoid cache conflicts between most frequent instruction functions. The former task is usually hard mainly because we must get a representative set of benchmarks and inputs and collect and merge obtained statistics from the different used benchmarks. Fortunately, our simulator provides an automatic way to perform this work: a profiling file may be passed as argument to indicate where to store and to accumulate the frequencies measured during a simulation. Once the set of benchmarks has been measured, the obtained profiling information may be used in turn to re-generate the optimized simulator.

In the other hand, the avoidance of cache conflicts is easily achieved by ordering execution functions according to the instructions profiles. So the functions of more frequent instructions would be positioned close enough to ensure they do not conflict according to cache sets. Table 4 sums up performances obtained with this approach that, unfortunately, gives very few improvements: in fact, conflicts in the host instruction cache are too infrequent to get significant benefit from this optimization.

Yet, the profiling information may be used to perform another kind of optimization. To take into account the branch predictor, we have to survey the branches involved in the instruction simulation. The execution is usually performed by getting a pointer on the function implementing the instruction and by calling it. From a general point of view, indirect pointer calls are badly processed by the branch predictor of the host machine, particularly in the case of a simulator where the branch targets change as often as instruction are different in the program. Therefore, most of these branches leads to bad prediction and big penalties on the simulation time. Although we have not been able to verify this statement as we do not know any profiling tool capable of tracing branch prediction misses, the improvement in the obtained results advocates for such a scenario.

```
if(id == ID_INST1)
        /* code INST1 */
else if(id == ID_INST2)
        /* code INST2 */
else if(id == ID_INST3)
        /* code INST3 */
…
else
        /* usual execution */
```

LISTING 2 Instruction Inlining

The solution to the previous issue is to avoid as indirect branches as possible. Using the instruction profiling described in the previous paragraph, it becomes possible to inline the action of more used instructions in the execution routine and to access them using a set of selections as in Listing 2. At simulation time, each selection gets its branch entry in the branch predictor (no interference between instructions) and the order in selection ensures the shorter time for the more frequent instructions.

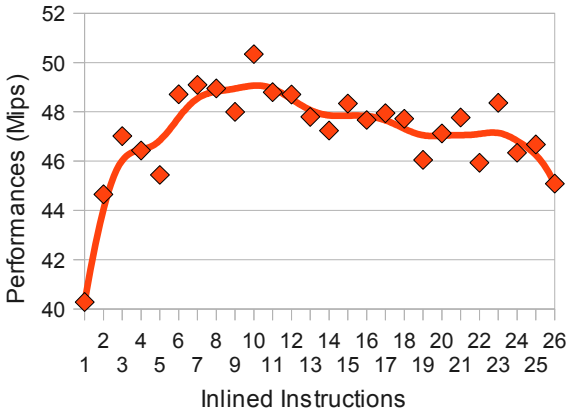

Figure 3: Effect of Instruction Inlining

To finalize this optimization, we have to determine how many instructions have to be in-lined. This may depend on the host machine and on the type of the simulated ISA as we get a double impact on the host branch predictor and on the host instruction cache. Experimental measurements, as shown in Figure 3 for PowerPC instruction set on Mälardalen benchmark [3], show that performances grow quickly until 6-8 instructions, become stable during a short phase and slowly decrease beyond 10-12 instructions. An interesting extension would be to provide an automatic way to find such a threshold.

Table 4 shows performances gain provided by the two methods to improve the execution step, REORG for execution

Table 4: Execution Optimization

	REORG		INLINE	
Benchmark	**(%)**	**(MIPS)**	**(%)**	**(MIPS)**
MINT	0.43	73,24	34,79	98,3
MFLT	6.3	52,95	10,68	55,13
AUTO	1.53	48,39	3,12	50,16
crafty	-0.72	63,77	-3.89	61,73
gcc	-3.03	60,77	-4.08	60,11
gzip	0.01	60,94	9,08	66,47
mcf	-3.47	57,32	0,69	59,79
parser	-0.32	53,46	-0.54	53,34
average	**0.09**	**58,98**	**6,95**	**63,13**

function reorganization according to the profile and INLINE for in-lined release. INLINE method exhibits an average speedup of 6,95% with an important peak of 34,79% for MINT. In fact, the profiling has been made on this benchmark and, accidentally, works also very well with gzip. Yet, this leads to an important issue of this technique: how to define a meaningful profile? Adversely, this allow also to tune precisely the simulator for a family of benchmarks that will be run many times, as done in hardware exploration.

This table show also the top performances, in MIPS, obtained by GLISS v2 on the selected benchmarks.

IV. Related Works

There are relatively little literature about the ISS optimizations we are aware of, except concerning Pre- or Just-In-Time native compilation of simulated programs [6][7]. These approaches give the best results but at the price of a loose of control on the simulation.

This lack of information is due to the fact that either some ADL are now developed in industrial context, or are not targeted to fast simulation. In this category, we include ADL as Expression [8], Lisa [9] or MIMOLA [10].

Harmless [11] is a relatively recent and active implementation of an ISS. Its ADL is relatively close to SNML but allows several trees of factorization for each attribute defining an instruction (image, syntax or action). This leads to a shorter description of the ISA description but at the price of a language and a description harder to learn and to handle. Whatever, Harmless has been successful to describe several ISA, RISC or CISC. The target generation language is C++ what makes relatively heavy the process of creating an instruction descriptor. Therefore, they have implemented an address-indexed cache, only direct-mapped, that prevents their performances from reaching ones of GLISS2.

From a perspective of hand-written simulators, there are few documented implementations. Nevertheless, we can site SimARM [12] that makes also use of decoded instruction cache but with an infinite size. As shown in our measurement, an infinite size cache may cause loss of performances, probably due to bad interferences in the host data cache.

V. Conclusion

This paper has presented the different techniques implemented in GLISS2 to improve the simulation speed, namely, (1) extreme specialization of the generated code, (2) optimization of the memory module, (3) use of an aggressive cache policy in the decoding step and (4) blocking of instruction descriptors in order to reduce the invocations of the decoder. These optimizations have been implemented keeping in mind the need of instruction level simulation and, as GLISS2 describes the ISA thanks to the NML ADL, they can be easily extended to any instruction set.

In future works, we plan to explore more deeply an important factor of slowdown, the simulation of floating-point operations. In the case of the PowerPC, this requires to support a complex set of bits in the float status register: (1) hard to extract from POSIX float support and (2) rarely accessed by real programs. Using libraries specialized into the host processor (often an x86), it may be possible to obtain more speedup. Our measurements have also shown that the optimization efficiency depends on the used benchmark. An interesting extension would be to provide a process to select the optimal simulator configuration according to the targeted benchmarks and even the actual host machine.

In addition, one may observe that NML works by in-lining big pieces of code, often only handling rare error cases, that, in turn, may have adverse effects in the instruction cache usage. It would be more valuable to isolate them in their own function. More generally, an effort should be done to optimize code generated from NML.

References

[1] T. Ratsiambahotra, H. Cassé, P. Sainrat, *A Versatile Generator of Instruction Set Simulators and Disassemblers*, International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2009), IEEE, p. 65-72, 2009.

[2] A. Fauth, J. Van Praet, and M. Freericks, *Describing instruction set processors using nML*, Proceedings of the 1995 European Conference on Design and Test, IEEE Computer Society.

[3] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[4] http://www.eecs.umich.edu/mibench/

[5] http://www.spec.org/

[6] I. Böhm, B. Franke and N. Topham, *Cycle-Accurate Performance Modelling in an Ultra-Fast Just-In-Time Dynamic Binary Translation Instruction Set Simulator*, *SAMOS'10, 2010.*

[7] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler, *Fast and Accurate Simulation using the LLVM Compiler Framework* , RAPIDO, 2009.

[8] A. Halambi, P. Grun, and al. *Expression: A language for architecture exploration through compiler/simulator retargetability*, European Conference on Design, Automation and Test (DATE), 1999.

[9] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Mey,. *Lisa - machine description language for cycle-accurate models of programmable dsp architectures*, DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation, 1999.

[10] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D.Voggenauer, *The mimola language version 4.1*, Technical report, Lehrstuhl Informatik XII, University of Dortmund, Dortmund, 1994.

[11] R. Kassem, M. Briday, J.-L. Béchennec, G. Savaton and Y. Trinquet, *Instruction Set Simulator Generation Using HARMLESS, a New Hardware Architecture Description Language*, IMCSIT'08, 2008.

[12] Alpa. Shah, *ARMSim: An Instruction-Set Simulator for the ARM processor*, Columbia University.

[13] S. Bhattacharya, *Generation of GCC Backend from Sim-nML Processor Description*, Master thesis, Indian Institute of Technology, Kampur, 2001.

[14] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, *OTAWA: an Open Toolbox for Adaptive WCET Analysis*, SEUS 2010.