High Speed Cycle-Accurate Processor Simulation Through Ahead of Time Compilation

Lovic Gauthier

National Institute of Technology, Ariake College
Department of Electronics and Information Engineering
150 Higashihagio-Machi, Omuta Fukuoka, 836-8585, Japan
Email: lovic@ariake-nct.ac.jp

Abstract— This paper presents techniques for implementing fast cycle-accurate processor simulators based on ahead of time compilation (AoT). AoT is usually assumed to suffer from a large compilation overhead, and is difficult to implement due to the dynamic behavior of some instructions. The paper explains how to overcome these issues and presents experiments with MIPS processor simulators showing that our approach can surpass state of the art methods and can simulate more than one billion clock cycles per second.

I. Introduction

Processor simulators are nowadays standard tools for embedded systems design. Allowing early verification, they significantly reduce the design cost, but traditionally suffered from very slow execution speed. In the past decade, techniques like binary translation (BT) [17] and just in time compilation/translation (JIT) [4] considerably increased their performance so that recent simulators like QEMU [5] are able to simulate tens to hundreds of millions of instructions per second. These techniques take root from compiled simulation [13] that simulates the execution of a binary program on a given processor by translating this binary into a C program. This program is then compiled and executed on the host computer. This method has given promising results but prove to be hard to apply on programs including branch instructions whose target addresses are known at run time only – in this paper, we call such instructions *dynamic branches*. Moreover, the compilation step of this approach considerably degrades its global performance.

This paper presents a set of techniques for overcoming the limitations with the dynamic branches, and shows the efficiency of persistent caching for overcoming the performance loss due to the compilation step. It also presents how the pipeline is simulated for achieving cycle-accuracy. These techniques are integrated into a framework which generates ahead of time compilation (*AoT*) processor simulators. AoT is a simulation method used in some virtual machines [15]. It includes all the translation, compilation and execution steps into an automatic flow. Our experiments showed that the generated simulators can execute about 1 billion cycles per second and surpass simulators based on JIT or BT techniques. The rest of the paper is as follows: section II gives some related works, section III explains the techniques used for high speed simulation, section IV gives some experimental results and related discussions before section V which concludes the paper.

II. Related works

Since the initial interpretation-based simulators, several techniques have been proposed for accelerating the execution so that recent simulators like [16, 6, 21, 20, 5, 3, 14, 8, 10, 11, 12, 18, 19], can achieve tens to hundreds of millions of simulated instructions per second.

In order to improve the performance of processor simulators, the most successful approach has been to convert the binary code which is to be executed on the simulated processor into native code for the host, and then to execute this native code. Compiled simulation was the first technique implementing this approach. This technique, presented in the introduction, suffers from limitations for handling dynamic branches and from compilation overhead. Hence, instead of generating a C program it has been proposed to directly generate host binary. This technique, called binary translation (BT) [17], proved to be more efficient than compiled simulation, but has the same weakness regarding the dynamic branches. To solve this, BT has been enhanced with Just-In-Time (JIT) [4] translation[1]. This latter technique translates the input binary as long as no dynamic branch is met, switches to interpretation mode when such branches are met, determines the target address, and switches back to BT mode from this address for a new translation and execution pass [6, 20, 5, 3, 11, 18]. A few other approaches mixed compiled simulation and JIT [7, 14, 8, 10, 19]. While achieving poorer results, such approaches are much simpler to implement, and can prove useful when a new processor simulator is to be implemented from scratch.

Compared with recent BT and JIT-based simulators, the approach presented in this paper is different in two ways: first it tackles the problem of the dynamic branches, from which it

[1] Such a combination is sometimes called dynamic binary translation.

is able to translate the totality of the input binary into a single program. Second, we assumed that the gain achieved by advanced optimizations of recent compilers, only applicable on high-level representations (IR) of the program (like static single assignment [9]), overcomes the compilation cost when combined with simple caching techniques.

The approach of paper [8] looks similar to ours since it also aims at handling dynamic branches without using JIT. However, in the details, their method is very different since it is actually close to disassembling the input binary. First it builds the control and data flow graph (*CDFG*) from the input binary, then it translates it to a C program that simulates it. With this approach, each function of the input binary is translated to a function in the simulation C program. Dynamic branches, which are often the consequence of calls to pointers to functions, are simulated alike: through pointer to function in the simulation program. While looking more natural than our direct translation approach, there are several difficult points. First, it is hard to extract a CDFG from the recent highly optimized input binaries. Especially, function calls and returns might be hard to detect since they can be, for optimization purpose, converted to standard jumps. Second, a CDFG is once again built and analyzed when compiling the simulation C code, which means that the same complex operations are performed twice. Third, C function calls and returns also include the stack management, yet the instructions implementing this management in the input binary must still be simulated. Once again, this approach ends up with twice the same operations, once for the input binary stack, and one for the host stack. Finally, some dynamic branches are not due to pointer to function, but either to switch/case statements, or arbitrary branches. If the former case is supported (through function calls), the latter is not.

III. The processor simulation

A. The global framework

Instead of designing a simulator for a specific processor, we designed a framework able to generate a processor simulator from a file describing the characteristics of each instruction, i.e., their machine code, their behavior at each stage of the pipeline and so on. The generated simulator then takes as input the program to execute, converts it to C code, compiles it to native code and launches the execution of this latter code. The details about the generation of simulators are beyond the scope of this paper and are therefore omitted for sake of concision.

B. Caching the compiled native code

Usually, during the design of an embedded system, the same program is simulated several times, either with different hardware configurations or with different input data sets. Hence, we added to the generated simulators a persistent cache that makes it possible to reuse previously compiled host code when subsequent simulation runs are to be performed. The implementation of this cache is straightforward: the compiled host code is saved as dynamic libraries whose file names are made of the input binary file name and the start address of the corresponding code chunk. Then, when simulating another time the same input binary, the C code generation and compilation steps are skipped and instead, the corresponding dynamic libraries are loaded. The input binary is still loaded, though, in order to fill the simulated processor's memory. Thereby, if a new data set is used, leading to a different binary, the cache is still used while the processor's memory content is valid. The main drawback of this simple approach is that if the code part of the input binary has been modified[2] the cache must be cleaned manually (change in the data does not require to clean the cache).

```
1  _8088:
2      /* add r1,r0,r2 */
3      r[1] = r[0] + r[2];
4      ...
5      /* j 8088 */
6      pc = 8088;
7      goto _8088;
```

Fig. 1. Implementation of a branch instruction

C. Implementation of the branch instruction

C.1. The general technique

Basically, the branch instructions are implemented in C using the goto statement. For that purpose there must be a label for each possible target of a branch instruction. In the simulator, such labels are named using the address they represent in the input binary. For instance, Fig. 1 shows the implementation of a branch back to address 8088. In the figure, line 3 is the implementation of an addition instruction where *r* is the array representing the general purpose registers. Then lines 6–7 are the implementation of the branch instruction: it sets the *pc* variable (representing the program counter) to the target address, and performs the actual branch with the goto of line 7.

This simple technique is sufficient if the target address is known at compile time, but is ineffective otherwise, i.e., with dynamic branches. For such cases we use the possibility given by recent C compilers to use goto on pointers to labels [1]. For that purpose, the simulator produces a lookup table of all the possible targets for a dynamic branch (cf. section C.2). Since the code is usually much smaller than the data, this table is implemented as an array whose index is the address in the binary code of the branch target. If the processor has some alignment restrictions, as it is the case for a majority of 16-bit and over processors, this index address is divided by the size of the alignment to reduce the size of the table. For instance, with the MIPS IV processor [2], the size of the table can be reduced by four. Fig. 2 gives an example of a label table, and the implementation of an instruction which branches to the address given by a register. In the figure, a portion of the table is given lines 2–8, where the unary operators `&&` are used to take the address of a label, and where `NULL` is the null pointer and indicates that no label is required for the corresponding address. The branch is implemented lines 12–18. Line 12 sets the program counter to the address given by register $1 (`r[1]`

[2]For instance, because of bug fixes.

```
1    /* Labels address table. */
2    static void* labels[] = {
3       &&_4096,   NULL,   NULL,   NULL,
4          NULL,   NULL,   NULL,   NULL,
5          NULL,   NULL,   NULL,&&_4140,
6          NULL,&&_4148,   NULL,&&_4156,
7          ...
8    };
9    void* label_ptr;
10   ...
11   /* jr r1 */
12   pc = r[1];
13   label_ptr = labels[(r[1]-4096)/4];
14   if (label_ptr) {
15      goto *label_ptr;
16   } else {
17      return jit_mode();
18   }
```

Fig. 2. Implementation of dynamic branches with pointer to labels

in the code), line 13 gets the label corresponding to this address[3] from the label table and line 14 checks the validity of the label. If it is valid, the branch is implemented by the goto of line 15. In the very unlikely case where a label is invalid, i.e., the corresponding entry in the table is NULL, the simulators enters temporarily in JIT mode (line 17 in the figure), to generate a new portion of C code where the required label is present. Such cases happen when the branch target detection described in the next section overlooked an address. Since the implementation of this JIT mode has no original feature and due to lack of room, its description is omitted in this paper.

C.2. Detecting the targets of the branch instructions

In theory, it is enough put a label before each simulated instruction in the generated C code for ensuring that each target address of branches corresponds to one label in the code. However, the resulting code will take more time to compile and will be far less optimized[4]. Actually, such a code is similar to code produced by a BT approach (cf. section IV.A). Hence, it is much better to insert the necessary labels and only them. To that end, our framework supports four cases: the branch is not dynamic, the branch is dynamic and implements a return from a function, it implements a call to a function pointer, or it implements a switch/case statement. Other cases are possible but are unlikely, especially if the input binary has been generated by a compiler. These unsupported cases are treated at simulation time in JIT mode as explained in the previous section.

The first case is straightforward since the target address is embedded in the branch instruction so that it is directly accessible at compile time. The second case is easy too, since a return instruction always branches to the instruction following a call (or one of the subsequent instructions, depending on the target processor). Therefore each instruction following a call (or one of its subsequents) has to be labelled. The third and forth cases are more complex to handle. For them it is necessary to consider how the compiler implements them. For the call to function pointer case, the target address is stored as a global value. For the case of the switch/case, the addresses of each switch block's statement are stored in a lookup table whose index is computed using the condition expression of the switch. For both cases, it is common for compilers to store these addresses in a section dedicated to the constant data in the resulting binary. For instance, gcc [1] stores them into the *rodata* section. Therefore, the possible target addresses of such dynamic branches can be found by scanning the relevant section of the input binary. However, other kinds of data are also stored into the section, e.g., the content of the string literals. In the framework, the number of false target addresses is reduced by discarding the values that do not correspond to addresses in the range of the executable code and ones that do not match the alignment requirements of the processor. In the experiments, this prove to be enough for removing a majority of such false addresses, since at worst, only five of them did remain.

D. *Cycle-accurate simulation*

In this paper, by cycle-accurate simulation, we mean that the simulated time in cycles when executing any portion of the input binary matches the one of the real processor. With this objective, it is not important to reproduce exactly the internal behavior of the processor, but instead it enough to ensure that the execution order and the time is valid after each instruction completes. This is trivial for a non-pipelined processor: after the simulation of each instruction, the cycle counter is simply increased by the number of cycles used by the instruction.

For a pipelined processor, an instruction completes at each cycle (or each fixed number of cycles) in a majority of the cases, but there may be variations due to stalls, or out of order instruction completions. These variations depend on the execution flow, hence we simulate them using flags for their detection and predicates for guarding the implementation of their actual effect. Depending on the simulated processor, the detection flags may be set when specific instructions are executed (e.g., branches), when resources are overused (counters are used for such a detection) or when accessed registers are not up-to-date. The latter case requires to associate a flag to each register: when an instruction writing to a given register is loaded, the corresponding flag is set, so that a further read access can be stalled, and unset when the register is updated. Such operations can severely slow down the simulation speed, but using our AoT approach, the optimizations of the compiler can discard them when they are not required[5], which is the most frequent case.

As an illustration, with the MIPS IV processor which has been used in our experiments, an instruction following a branch instruction is executed before the branch is taken. Fig. 3 shows the management of the pipeline for such a branch instruction. In the figure, a branch instruction is located line 2 where the `jmp` flag is set to one. This branch is actually taken lines 8–10, i.e., after the following instruction located line 5 is executed. For that purpose, `jmp` is checked line 7 to decide if the branch is to perform or not. This way, if an execution flow goes to address `7096` (line 3 in the code) without passing through the instruction of line 2, the code of lines 8–10 is not executed, which matches the real processor's behavior.

[3]The address is converted to a label index by removing the code's starting address and dividing the result by four since instructions are all 4 byte long.

[4]Since there will be more basic blocks.

[5]For instance if there is no dependency among some instructions.

```
1       /* j 8196 */
2       jmp = 1;
3    _7096:
4       /* add r1,r0,r2 */
5       r[1] = r[0] + r[2];
6       /* Execution of j */
7       if (jmp == 1) {
8           pc = 8196;
9           jmp = 0;
10          goto _8196;
11      }
```

Fig. 3. Example of the handling of pipeline effects

```
1  _4120:                                ; _4120:
2     movl    -164(%rbp), %r13d          ;   r[3] = r[0]+65535;
3     decl    %r13d
4     movl    %r13d, -164(%rbp)
5     incl    %eax                       ;   count++;
6  _4124:                                ; _4124:
7     movl    $131072000, -168(%rbp);    ;   r[2] = 2000 << 16;
8     incl    %eax                       ;   count++;
9  _4128:                                ; _4128:
10    movl    -160(%rbp), %r15d          ;   mem[r[2]+0] = r[3];
11    movq    -168(%rbp), %rcx
12    movl    %r15d, (%rdx,%rcx,4)
13    incl    %eax                       ;   count++;
```

Fig. 4. Example of code produced in BT mode

IV. Experiments

A. Setup

In order to compare the performance of the proposed approach with BT and JIT-based approaches we enhanced our framework with support of such techniques. The goal was to achieve a fair comparison: apart from the core techniques, several implementation details might cause strong variations in the performances. The framework already includes a JIT mode (cf. section III.C), the generated simulator is hence configured to be in this mode all the time for implementing a JIT-based approach. However, including a full implementation of BT would have required to completely change the processor simulator generation core of the framework. Instead, we simulated the BT mode by generating C code whose compilation result is equivalent to what would be the result of a real BT. The idea is that BT should be equivalent to compiling a program where each basic block corresponds to the simulation of one instruction. Hence we added to the framework a mode where such a C code is generated (basically, this code includes a label for each simulated instruction, and dummy branches that prevent the basic blocks from being merged or removed by the optimizations). Fig. 4 shows some assembly code produced by the BT mode (after being compiled), which is actually very close to what we produced by hand when studying BT. For sake of understanding, the corresponding C code is given as comments on the right of the assembly code (*count* is the clock cycle counter). Note that when measuring time with the emulated BT approaches, we removed the compilation time to match an actual BT approach.

This modified framework has been tested with the combinations of techniques given in table I. In the table, *Name* is the name of the configuration, *AoT/JIT* indicates if our AoT approach is used or if JIT is used, *C/BT* indicates if C is generated then compiled or if BT is used, and *First/Next* indicates if it is the first simulation run (i.e., the simulation cache is empty and misses will happen) or if it is a subsequent one

TABLE I
The simulator configurations

Name	AoT / JIT	C / BT	First / Next run
JIT_BT	JIT	BT	-
AoT_C_f	AoT	C	First
AoT_C	AoT	C	Next
AoT_BT	AoT	BT	-
JIT_C_f	JIT	C	First
JIT_C	JIT	C	Next

(i.e., simulation cache miss does not happen). Among the configurations, AoT_C is the approach proposed in this paper — AoT_C_f being this configuration used for the first simulation run — and JIT_BT is the one used in recent simulators like [6, 20, 5, 3, 11, 18]. In order to see the advantage of each specific technique, AoT_BT and JIT_C have also been tested.

The applications used for the experiments are taken from the Dhrystone (*dhry* in the graphs' results), and the MiBench (the other applications) benchmarks. Each application includes a few dynamic branches: from 0.001% of the executed instructions for *crc* (a MiBench application) to 3% for *dhry*, and 1% on average. In addition, since the simulations are extremely fast, the input data have been enlarged for several applications in order to achieve measurable execution times.

Finally, the compiler used for generating the native code is gcc 4.2.1 and the host is a MacBook Pro computer equipped with a 2.5Gz Intel Core i5, 8GB of RAM and 375 GB of SSD.

B. Results

Fig. 5 gives the execution speeds of each approach for each application. As it can be seen in the figure, using JIT is less efficient than using our AoT approach, which confirms the merits of the proposed techniques for handling branches. This result is not surprising since the cost of dynamic branches includes not only the change of execution mode, but also the requirement of compiling or loading a new portion of code and the lack of optimizations induced by the fragmentation of the simulation code. The difference between the first and the second run is also very large, which shows the importance of caching, since without it, using AoT_BT (our AoT approach combined with BT) prove to be better in a majority of the cases. On the contrary, AoT_C surpasses all the other approaches in the second run for all the cases. In order to analyze more finely the respective performance of the approaches, Fig. 6 gives the relative execution time of each step of the simulation process. In the figure, *load* is the time for loading the input binary file from the host's disk to the simulated processor's memory, *translate* is the time for translating it to C (or the target binary for BT) and detecting the target addresses of the dynamic branches, *compile* is the time for compiling the C or loading the already compiled/generated host binary, and *simulate* is the time for the actual simulation. For sake of readability, only AoT_C and AoT_BT are shown (the JIT-based approaches have about the same figures, apart from the simulation time which is larger). As seen in the figure, the load time is sometimes very large compared to the simulation time (e.g., *rijndael*). This is due to the important size of the input data set of some applications.

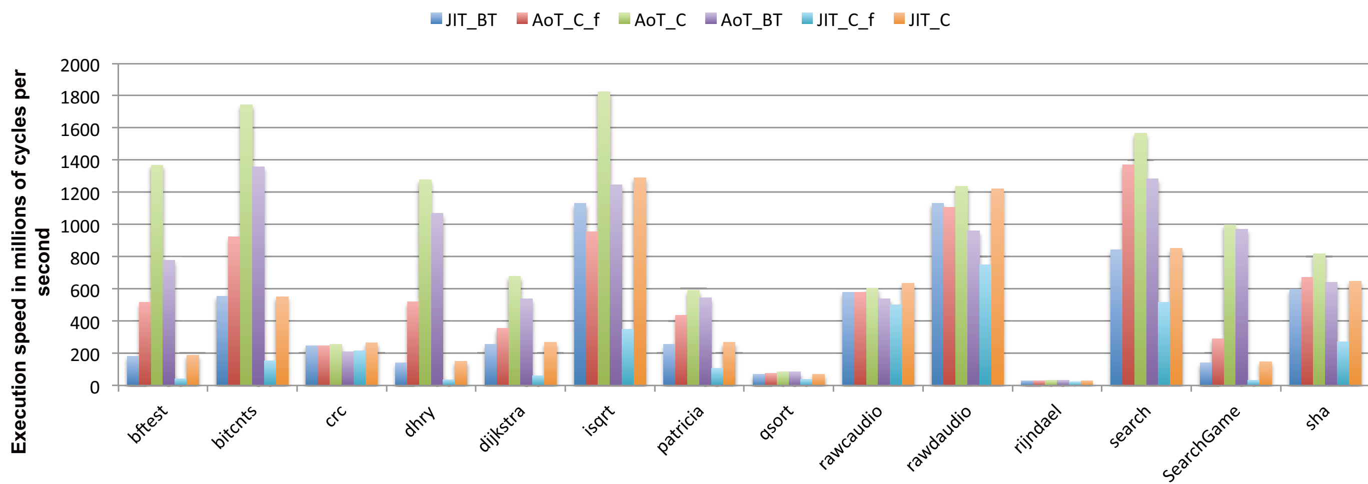

Fig. 5. The execution speeds of the different simulation approaches

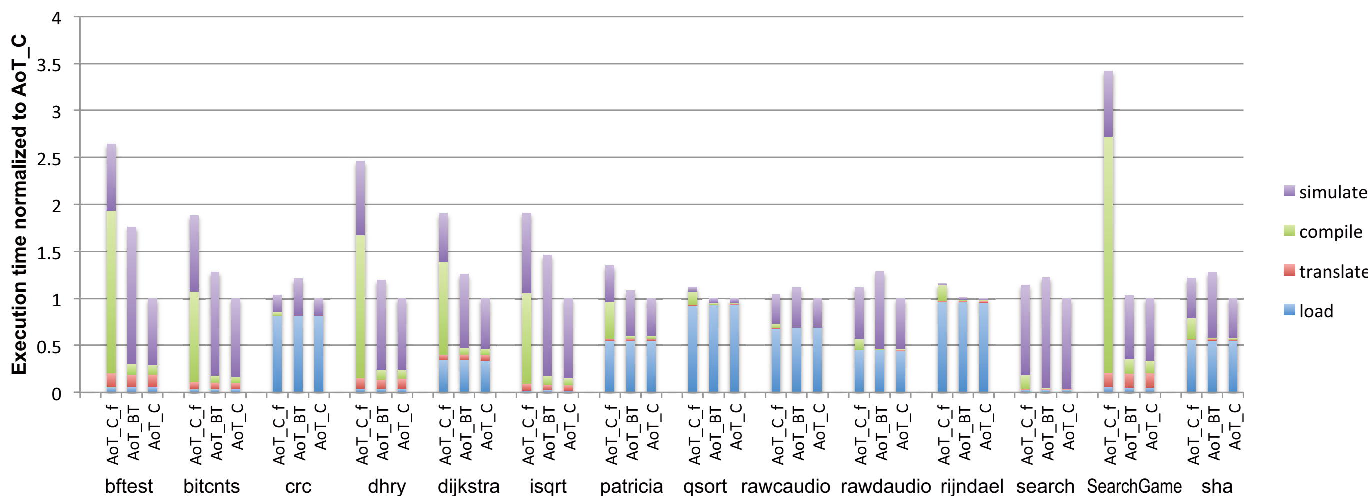

Fig. 6. The detailed execution times of the different simulation approaches

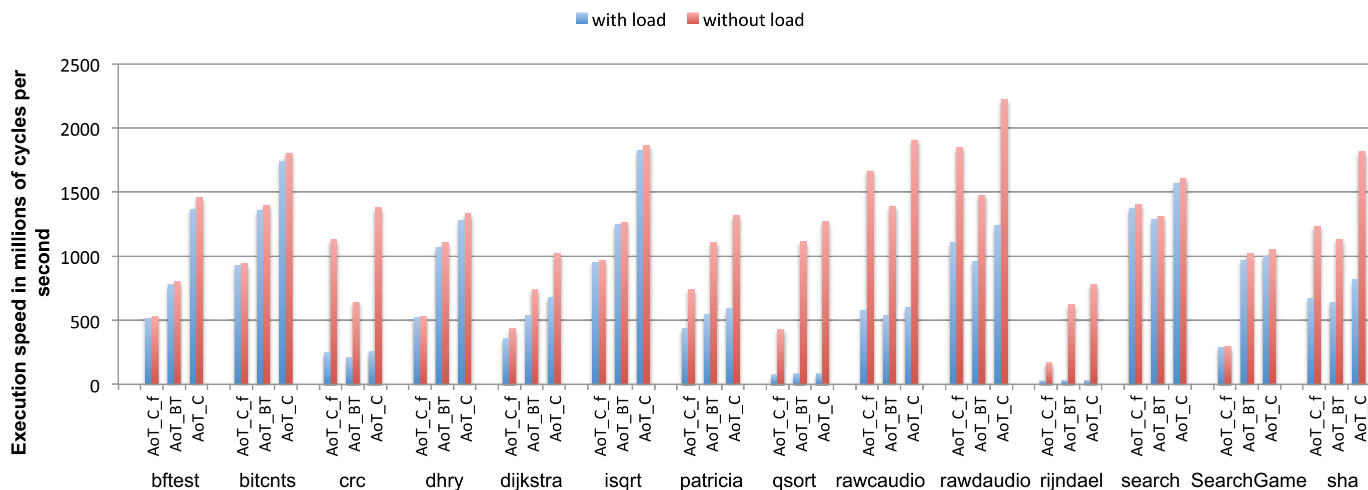

Fig. 7. The simulations speeds with and without the file loading time

Real-world applications (not formatted for a benchmark) taking their input data from some input devices should not be that long to load. By contrast, the translation time is usually negligible. More realistic results may be obtained by removing the load time. This is shown in Fig. 7, where the execution speed is given, with and without the load times, for AoT_C and AoT_BT. This last figure shows that, both approach are very fast, but, without the load time, AoT_C is on average 40% faster than AoT_BT. This confirms that the C compiler's resulting code is more efficient than the one achieved by BT.

V. Conclusion

This paper presented an approach for producing very fast cycle accurate processor simulators using ahead of time compilation. Several techniques are introduced for supporting dynamic branches and pipeline, while persistent caching is used for reducing the compilation time among several simulation runs. Experiments showed that a MIPS IV processor simulator generated by our framework can execute more than one billion cycles per second and surpasses state of the art ones based on just in time compilation or binary translation.

As future work, we plan to see how to speed up the simulation of the communication with peripherals in order to produce very fast simulators for complete embedded systems.

References

[1] GCC, the GNU compiler collection. `https://gcc.gnu.org/`.

[2] MIPS IV instruction set. `http://www.weblearn.hs-bremen.de/risse/RST/docs/MIPS/mips-isa.pdf`.

[3] B. Alexander, S. Donnellan, A. Jeffries, T. Olds, and N. Sizer. Boosting instruction set simulator performance with parallel block optimisation and replacement. In *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*, ACSC '12, pages 11–20, Darlinghurst, Australia, Australia, 2012. Australian Computer Society, Inc.

[4] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[6] I. Bohm, B. Franke, and N. Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 1–10, July 2010.

[7] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(12):1625–1639, Dec 2004.

[8] Moo-Kyoung Chung and Chong-Ming Kyung. Improvement of compiled instruction set simulator by increasing flexibility and reducing compile time. In *Rapid System Prototyping, 2004. Proceedings. 15th IEEE International Workshop on*, pages 38–44, June 2004.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[10] J. D'Errico and Wei Qin. Constructing portable compiled instruction-set simulators-an adl-driven approach. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, March 2006.

[11] D. Lockhart, B. Ilbeyi, and C. Batten. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing jit compilers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 256–267, March 2015.

[12] Mingsong Lv, Qingxu Deng, Nan Guan, Yaming Xie, and Ge Yu. Armiss: An instruction set simulator for the arm architecture. In *Embedded Software and Systems, 2008. ICESS '08. International Conference on*, pages 548–555, July 2008.

[13] C. Mills, S. C Ahalt, and J. Fowler. Compiled instruction set simulation. *Software-Practice and Experience*, 21(8):877–889, 1991.

[14] Wai Sum Mong and Jianwen Zhu. Dynamosim: a trace-based dynamically compiled instruction set simulator. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 131–136, Nov 2004.

[15] A. Nilsson and S.G. Robertz. On real-time performance of ahead-of-time compiled java. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 372–381, May 2005.

[16] M. Reshadi, P. Mishra, and N. Dutt. Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation. *ACM Trans. Embed. Comput. Syst.*, 8(3):20:1–20:27, April 2009.

[17] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, February 1993.

[18] T. Spink, H. Wagstaff, B. Franke, and N. Topham. Efficient dual-isa support in a retargetable, asynchronous dynamic binary translator. In *Embedded Computer Systems: Architectures, MOdeling and Simulation, SAMOS XV. International Conference on*, June 2015.

[19] A. Trouvé and K. Murakami. Ffast: Efficient application of compiled simulation techniques to a fast iss over a virtual machine. In *Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO2010*, pages 23–34, January 2010.

[20] Tse-Chen Yeh, Guo-Fu Tseng, and Ming-Chao Chiang. A fast cycle-accurate instruction set simulator based on qemu and systemc for soc development. In *MELECON 2010 - 2010 15th IEEE Mediterranean Electrotechnical Conference*, pages 1033–1038, April 2010.

[21] Jianwen Zhu and D.D. Gajski. An ultra-fast instruction set simulator. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 10(3):363–373, June 2002.