Fast Translated Simulation of ASIPs

Zdeněk Přikryl, Jakub Křoustek, Tomáš Hruška, and Dušan Kolář

Brno University of Technology, Faculty of Information Technology
Božetěchova 2, 612 66 Brno, Czech Republic
{iprikryl, ikroustek, hruska, kolar}@fit.vutbr.cz

Abstract

Application-specific instruction set processors are the core of nowadays embedded systems. Therefore, the designers need to have powerful tools for the processor design. The tools should be generated automatically based on a processor description. One of the most important tools is the simulator. It is used during a testing phase of the processor design and during target software development. The key feature of the simulator is its speed. The concept of a special simulation type – translated simulation – is presented in this paper. This simulation exploits information from a target C compiler. Both the simulator and the C compiler are generated based on the processor description in an architecture description language ISAC. Experimental results of this concept show very good simulation speed and fast generation of the simulator.

Keywords and phrases Hardware/sofware co-design, simulation, architecture description languages, application-specific instruction set processors.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.93

1 Introduction

Embedded systems have become essential part of our nowadays lives. One can find them almost everywhere. There can be one or more application-specific instruction set processors (ASIPs) inside an embedded system. Each processor has usually dedicated functionality and it is highly optimized for it. There are many trade-offs among which part of functionality should be implemented directly in the processor and which part should be implemented in software. The process of optimal solution searching is called design space exploration (DSE). Therefore, the designer should have a good integrated desktop environment (IDE) for the processor design. The IDE should provide automatic tool-chain generation based on the processor description. The tool-chain consists of the tools for processor programming, such as an assembler or C compiler, and of the tools for processor simulation, such as a simulator or profiler.

The processor itself can be described using either hardware description language (HDL) or architecture description language (ADL) (see [11]). Generally, ADLs are better for fast DSE and rapid processor prototyping, since ADL hides hardware details. Those details can be unknown at the beginning of the processor design or the designer does not want to take care of them.

One of the tools used during the whole processor design is a simulator. Therefore, the simulator has to be fast enough. Furthermore, the simulator is also used for the target software development (often at the same time as the hardware is designed – hardware/software co-design). There are several different types of simulators. Each of them is usually used in different phase of the processor design (less accurate simulator during the first steps in DSE, more accurate simulator during the preparation of final hardware realization). Various types are discussed in the section 2, and advantages or disadvantages are highlighted.

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 93–100

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our project running at Brno University of Technology is called Lissom (see [7]). It provides whole IDE for the processor design and tools for multiprocessor system on a chip design. In this paper, we present the concept of a block-accurate simulator called *translated* simulator. It uses additional information of basic blocks from the target C compiler. It is based on the LLVM (Low Level Virtual Machine) platform [8]. The compiler can be also generated based on the processor description using the ISAC (Instruction Set Architecture C) language. The ISAC language is inspired by the LISA (Language for Instruction Set Architectures) language [5] and it has been developed within the Lissom project.

2 State of the Art

There are a few projects which try to give the developer a whole IDE for the processor design. Each of them uses its own description language which has been developed within the project. An open source project ArchC [1] uses ADL called ArchC. It is a description language for pipeline systems based on SystemC. The processor description is composed of several parts. The designer can describe resources, such as memories or registers, instruction set and its behavior. The behavior is described with SystemC functions in shared libraries.

Another widely used ADL is LISA. The processor description in LISA language is composed of several parts. In one part, resources are defined. In the other part, an instruction set with behavior and processor microarchitecture is described. In both projects (ArchC and LISA), the interpreted and compiled simulators are available, but none of them supports translated simulator.

At the Vienna University of Technology, an ADL called xADL (see [2]) was developed. The processor is described with hardware blocks which are interconnected. The xADL language supports generation of the translated simulator. The LLVM platform is used for simulator creation; therefore, the creation of simulator takes a long time since the whole LLVM has to be compiled.

An introduction to the simulator terminology used in this paper is in the following text. The basic type of simulator is an interpreted simulator. The run of interpreted simulator is based on the following concept. It fetches an instruction then it decodes the instruction, and executes it. Therefore, it is not dependent on simulated application and it allows self-modifying code out of the box. On the other hand, it constantly fetches and decodes the same instructions (e.g. instructions within a loop). Hence, this slows the simulator down.

Another type is a compiled simulator. Unlike the interpreted simulator, the compiled simulator is created in two steps. In the first step, a simulated application is analyzed. In the second step, based on the analysis, the simulator itself is created. It is clear that the basic type of compiled simulator cannot simulate self-modifying code and it is dependent on the analyzed application. Note that this is not true in the case of dynamic compiled simulators (see [12]).

Other important feature of simulators is the simulator accuracy. Basically, the simulator can be cycle-accurate, instruction-accurate or block-accurate. In the first type, the basic step of the simulation is single clock cycle. Therefore, this type of simulator is very close to hardware and gives the most relevant information about the behavior of a real processor. On the other hand, the speed is not very good, since the whole microarchitecure is simulated. Therefore, this type of simulator is used when the processor design is stable enough.

In the second type, the basic step of the simulation is single instruction. The processor microarchitecture is not simulated. This type is used for target software development (i.e. the software which will run on designed processor), or it can be used for virus detection where

an instruction-accuracy is enough. Note that the cycle-accurate and instruction-accurate simulators can be either interpreted or compiled [12].

The block-accurate simulator uses a whole basic block in a simulated application as a basic step of the simulation. The basic block is an indivisible sequence of instructions with one entry point (start address), one exit point (end address), and no branch instructions within it. These addresses cannot be always determined during static analysis of a simulated program. There can be (and in a real processors usually is) a branch instruction, which gets a destination address from a register. Therefore, this address is known only during a simulation. The start and end addresses of all basic blocks are known during a compilation, so the compiler can save this information for further usage by simulator. Since we need to preprocess this information, the block-accurate simulators are only compiled; therefore they are dependent on particular simulated application.

3 ISAC Language

The ISAC language falls into so-called mixed architecture description languages. It means that the processor instruction-set with processor microarchitecture is described in one model. The processor model consists of two parts in the ISAC language. In the first part, the processor resources, such as registers or caches, are described. In the second part, processor instruction-set together with microarchitecture is described. The basic construction of the second part is *operation* construction. The operation can have several sections. The section describes either instruction-set or microarchitecture and forms one of the four basic models of processor. Each model describes the processor from different point of view. The models are: *instruction-set* model, *timing* model, model of *instruction analyzers hierarchy* and *behavioral* model.

The instruction-set model is formed by the *assembler* and *coding* sections. The assembler section describes the textual form of an instruction (assembly language). The coding section describes the binary form of the instruction (machine code). The timing model is formed by the *activation* section. This section denotes what and when is done in the microarchitecture of processor (e.g. timing of processor pipeline). The model of instruction analyzers hierarchy is formed by the section *structure*. This section describes timing of instruction decoding. The behavior model is specified by sections *expression* and *behavior*, where the ANSI C language is used (i.e. ANSI C descibes the behavior of instructions and processor microarchitecture). Note that the expression section has the same meaning as the return statement in a function (i.e. it is used for returning of a value if particular operation is used during instruction decoding).

Operations can be grouped according to some criteria, such as similar functionality (e.g. operations describing arithmetic instructions). The *group* construction is used for grouping of that operations or other groups. An operation can use other operations or groups using the *instance* statement (e.g. an operation describing move instruction uses another operation describing immediate operand). The processor model consists of a resource description and many operations and groups. There is one mandatory operation called `main`. This special operation is used for synchronization (i.e. clock cycle generation). An example of two operations is in Listing 1. There is an operation describing 8-bit immediate operand using 8-bits attribute and other operation describing `move_acc` instruction. The second operation uses results from previous operations (i.e. it uses results from expression sections which is the value of immediate operand). More information can be found in [9].

Basically, the processor can be described on instruction-accurate or cycle-accurate level

Listing 1 Example of ISAC Language Source Code

```
// Operation with one attribute
// attr for 8 bit operand
OPERATION imm8 {
  ASSEMBLER { attr=#U };
  CODING { attr=0bx[8] };
  EXPRESSION { attr; };
}
// Operation describing move
OPERATION move_acc {
  INSTANCE imm8;
  ASSEMBLER { "move_acc" imm8 };
  CODING { 0b0101 imm8 };
  BEHAVIOR { acc = imm16; };
}
```

by the ISAC language. Note that the processor model at instruction-accurate level has operation `main` with the structure section.

4 Concept of Translated Simulation

The following notation is used in this section. A target C compiler is the generated C compiler. It is generated from the processor model and it is based on the LLVM platform. A target application is the application which will run on the designed processor. A host C compiler is *gcc* compiler which compiles the generated simulator itself. The process of the simulator generation has three parts. The processor description has to be on the instruction-accurate level. The first part is performed only once for any particular processor description. The next two parts are target application specific, so they have to be performed every time when the target application is changed.

In the first part, the analyzer of the target application is generated. It is generated only once and it is based on the processor description (i.e. it does not have to be re-generated until the processor description has changed). This analyzer is similar to the disassembler, so it accepts an application in the machine code. But instead of emitting the assembly code, it emits C code. The analyzer itself is based on the enhanced formal models *coupled finite automata* [6] and *lazy finite automata*. *Lazy finite automaton* M is septuple $M = (Q, \Sigma, \delta, s, F, S, z)$, where Q is a finite set of states, $\Sigma = \{0, 1\}$ is an input alphabet, $s \in Q$ is a starting state, $F \subseteq Q$ is a set of final states, $\delta = Q \times \Sigma^* \times Q$ is a finite transition relation, S is a set of *semantic actions*, and z is a relation $z \subseteq \delta \times S$. The relation z assigns semantic actions to transition relations. The semantic action is indivisible sequence of a C code which is executed when a particular transition is taken. Definitions of configuration, move, and accepted language are analogical to definitions in normal lazy finite automaton.

Coupled finite automata C used in the analyzer is a triple $C = (M_1, M_2, h)$, where M_i is a lazy finite automaton for $i = 1, 2$, and h is a bijective mapping from δ_1 to δ_2. Definition of bijective mapping h, and translation by coupled finite automata are analogical to definitions in normal coupled finite automata (see [6] for more details). The automaton M_1 is used as an instruction parser ($\Sigma_1 = \{0, 1\}$) and the automaton M_2 is used as a C code generator. The set S_2 contains the modified C code from the behavior and expression sections, which are taken from the processor description. The content of the behavior section is changed in a way that the constants, which are represented by attributes, are replaced by their evaluation (values

are obtained from the automaton M_1 during translation). Furthermore, each statement is encapsulated, so the C code is only printed into a file, not executed. Let's assume the behavior section from the Listing 1. In the simple case, the content of original expression section of the *imm8* operation, `attr;`, is changed to `fprintf(fp, "imm8 = %d\n", attr);`, and the content of original behavior section of the *move_acc* operation, `acc = imm8;`, is changed to `fprintf(fp, "acc = imm8;");`.

In the second part, the core of translated simulator is created (i.e. the analyzer generates a C code based on the target application). The generated output C code has to be organized somehow. If the output would be only one single function, then, in the the case of large target application, the function would become uncompilable (e.g. problems with optimizations, problems with virtual memory, etc.). Therefore, the address space of the designed processor is divided into so-called segments. Each segment has the same fixed size, which is set during the creation of an analyzer by the developer. The size has to be equal to some power of two (e.g. 512 or 1024). The reason for that action is explained later. Functions are generated for each segment. It simulates instructions within the segment. This function has one parameter. It is used for passing the program counter. Each function contains single `switch` statement which takes this parameter. The `case` statement is generated for each instruction within the segment. Note that the `case` bodies are generated by the M_2 automaton. In a straight approach, each `case` is ended with a `break` statement. There are two main reasons why this approach does not allow effective host compiler optimization. Firstly, each `break` ends the function. That means that the computed values, which can be used in the next simulated instruction, are swapped out from the host registers to the main memory. From the host processor point of view, it would be better to keep these values in the registers. Secondly, the `case` does not allow additional optimizations since it creates the end of basic block in the simulator code. The side effect of the two mentioned constructions leads to worse cache hit/miss ratio too. Therefore, the following improvement is used.

Since the analyzer knows the starting and ending addresses of basic blocks in the target application (they are stored as debug info in the target application), the `break` is generated only if an address of an analyzed instruction is equal to an ending address of some basic block. Otherwise, the simulation of new clock cycle is performed (i.e. the behavior section of `main` operation is executed). The `case` is generated only if an address of an analyzed instruction is equal to a starting address of some basic block. The Fig. 1 shows previous principle. Each instruction has 32 bits and the address space can be addressed by 8 bits in

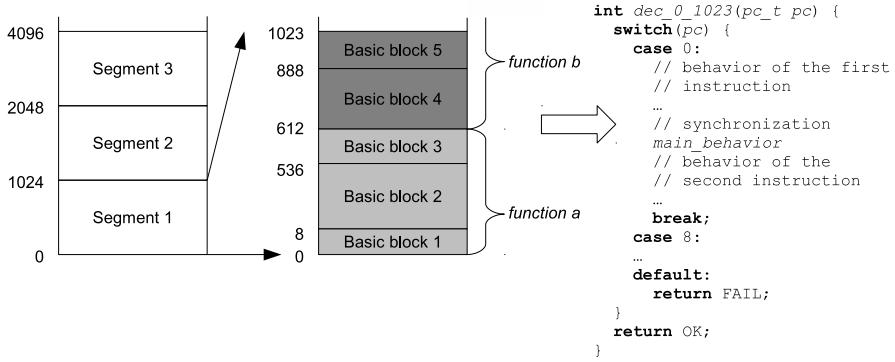

Figure 1 Principle of a translated simulator generation

this example. Note that there is no `break` for address 0 (it is not ending address) and there it no `case` for the address 4 (it is not starting address). Therefore, the unmodified behavior section of synchronize operation `main` is generated there.

The whole target application is represented by several functions. These functions are stored in a table. The key to this table is created by the right bit shift of an instruction address value. The count of bits needed for shifting is computed from the segment size (square root of segment size). The limitation of the segment size (power of two) guarantees a fast transformation from the addresses to the keys used in the table. Note that the valid addresses are addresses of the start and end of basic blocks. The simulator itself is formed by a loop which calls particular functions from the table together with the execution of the unmodified behavior section of synchronize operation `main`.

In the third part, the simulator itself is created via a compilation of target application independent parts, such as the representation of the resources, and target application dependent part (i.e. functions generated by the analyzer).

5 Experimental Results and Future Research

Several experiments of translated simulation concept were performed. As testing processor architectures we chose MIPS and VEX. Both processors are described on instruction-accurate level in the ISAC language. MIPS is a 32bit RISC (reduced instruction set computer) architecture developed by MIPS Computers Systems. The instruction-set of MIPS is in version MIPS32 Release 1. VEX is a four-slot 32bit VLIW (very large instruction word) architecture designed by HP [4]. Each slot is unique (i.e. each slot processes different types of instructions).

MiBench test suite [10] was used for testing and simulation speed measuring. All simulations were performed on the same host – Intel Core 2 Quad with 2.8 GHz, 1333 MHz FSB and 4GB RAM running 64-bit Linux based operating system. The gcc (v4.4.4) compiler with optimizations (–O3) was used for creation of simulator generators and simulators. All results are the average values of several runs of each test (differences of values from average are in tenths of a percent).

Fig. 2 shows the performance comparison of all simulator types for MIPS and VEX. As we can see in this figure, the speed of translated simulation is approximately 70% faster than compiled simulation and up to four times faster than interpreted simulation.

In average, the times needed for creation of compiled simulator generators are 5.03s for MIPS and 17.42s for VEX. The creation of translated simulator generators takes 4.93s

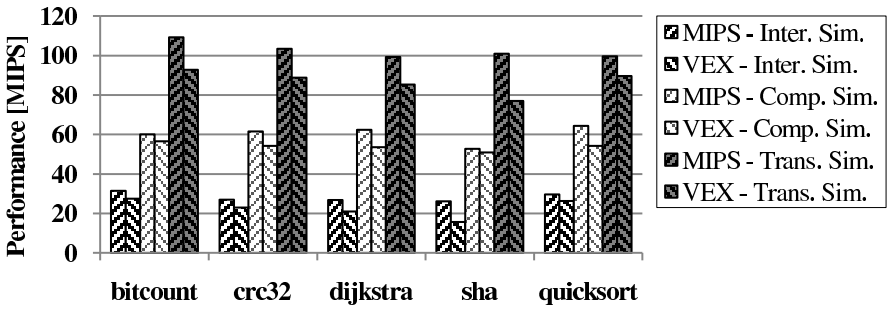

Figure 2 Performance comparison of all simulator types

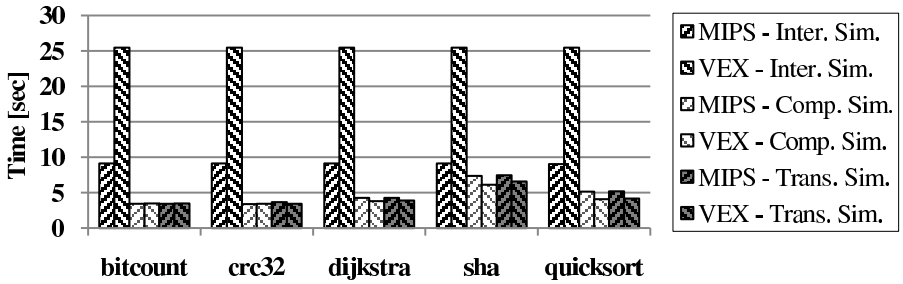

Figure 3 Simulator generation time

for MIPS and 17.17s for VEX. In Fig. 3, we can see generation times of simulators based on a target application (for the compiled and translated simulation). The time needed for generation of an interpreted simulator is constant because it is application independent. The sum of times needed for creation of translated simulator generator and simulator itself is lower than creation time of interpreted simulator, based on target application complexity. This is another advantage of this simulator type.

Our concept of translation simulation is fully competitive. For example, our solution is in average 40% faster than the concept of translated simulation created at Vienna University of Technology (according to results in [3]). The comparison was made on MIPS architecture and the set of five MiBench algorithms.

6 Conclusion

The concept of translated simulation is presented in this paper. The simulator is generated based on a processor description and a target application. A processor is described using the architecture description language ISAC. The generator of simulator needs to know all starting and ending addresses of all basic blocks in the target application. This information is obtained from the C compiler. It is based on LLVM platform and it is generated from the same processor description. The generator of a simulator is based on several formal models. The same formal models are also used in other generators, such as hardware description generator. Hence, no additional huge verification of hardware realization is needed. The experimental results show very good simulation speed and the time needed for a creation of the simulator itself is low. All mentioned features provide the powerful platform for ASIP and target software development.

Acknowledgements This work was supported by the research funding MPO ČR No. FR-TI1/038, BUT FIT grant FIT-S-10-2, doctoral grant GA ČR 102/09/H042, SMECY, and by the Research Plan No. MSM 0021630528.

References

1 ArchC Architecture Description Language. http://archc.sourceforge.net/

2 Brandner, F.: Compiler Backend Generation from Structural Processor Models. PhD thesis, Institut für Computersprachen Technische Universität Wien (2009)

3 Brandner, F.: Fast and Accurate Simulation Using the LLVM Compiler Framework. In RAPIDO '09: 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (2009)

4 Fisher, J. A., Faraboschi, P., Young, C.: Embedded Computing – A VLIW Approach to Architecture, Compilers, and Tools. Morgan-Kaufmann Elsevier Publishers (2005) 978-1-55860-766-8

5 Hoffmann, A., Meyr, H., Leupers, R.: Architecture Exploration for Embedded Processors with LISA. Kluwer Academic Publishers (2002) 4020-7338-0

6 Hruška, T., Kolář, D., Lukáš, R., Zámečníková, E.: Two-Way Coupled Finite Automaton and Its Usage in Translators. In: New Aspects of Circuits, Heraklion, GR, WSEAS (2008) 445-449

7 Lissom Project. http://www.fit.vutbr.cz/research/groups/lissom/

8 LLVM Compiler Infrastructure. http://llvm.org/

9 Masařík, K.: System for Hardware-Software Co-Design. FIT BUT, Brno, CZ (2008)

10 MiBench test suite. http://www.eecs.umich.edu/mibench/

11 Mishra, P., Dutt, N.: Processor Description Languages. Morgan Kauffman Publishers (2008) 978-0-12-372487-2

12 Přikryl, Z., Hruška, T., Masařík, K., Husár, A.: Fast Cycle-Accurate Compiled Simulation. In PDeS '10: 10th IFAC Workshop on Programmable Devices and Embedded Systems (2010)