Automatic Generation of a Simulation Compiler by a HW/SW Codesign System

Hideaki Yanagisawa, Minoru Uehara, Hideki Mori
Information and Computer Sciences
Toyo University, Japan

mist@mo.cs.toyo.ac.jp, uehara@cs.toyo.ac.jp, mori@cs.toyo.ac.jp

Abstract

An ISA (Instruction Set Architecture) simulator is an indispensable tool for the design, development and customization of new processors. ISA simulators provide various features such as checking instruction behavior and/or estimating performance and chip size for a target processor. Due to the increasing complexity of chip architecture and time to market pressure, performance becomes one of the most important features for ISA simulators. This paper describes a technique for automatic generation of a simulation compiler that translates a target code into a host code, by a HW/SW codesign system. We proposed an indirect approach for the simulation compiler; whereby, an execution file of a target processor is modeled in the C language and translated into the execution file of the host machine using a compiler. An advantage of our approach is the ability of the compiler on the host machine to easily translate a target code into host codes.

1. Introduction

An ISA (Instruction Set Architecture) simulator is an essential tool for developing new processors. ISA simulators have many features to enable functions such as the verification of instructions behavior, application programs and internal status, the evaluation of architecture (e.g. construction of pipeline stage, amount of register and bit width), and estimations of the performance of application programs and the cost of hardware.

Especially, in the development environments of an ASIP (Application Specific Instruction set Processor) that is implemented in embedded systems, an ISA simulator is required for rapid prototyping because an ASIP changes the architecture and the combinations of instructions, depending on the application programs.

Many ISA simulators adopt an interpretive simulator because of its flexibility. However, the performance of an interpretive simulator is very slow; and because of increasing time to market pressure, performance has become an important factor for ISA simulators. To rapidly design and develop a new processor, a high

performance ISA simulator is required. Depending on its purpose, each ISA simulator has a different combination of functions. One possible solution to the performance problem may be dividing up a simulator's roles such as estimation, performance analyzing, and behavior and architecture checking; while another solution could be dividing simulation time into compilation time and simulation time using a simulation compiler. Simulation compilers translate target code into host code directly; however, direct translation approaches are not portable because they depend on hosts.

To achieve the goal of higher simulation performance with flexibility and portability, we propose an indirect approach for generating an ISA simulator; one that is a compiled simulator. In our approach, the object codes of the application programs for a target machine are translated to a simulator which consists of a description of the instruction behavior and the assembly code of the application in the C language. Then the simulator in C is translated to the execution file by the compiler on the host machine.

To model instructions for a target processor is a tedious and error-prone process. To archive the goal of rapidly arriving at a prototype, we generate the indirect simulation compiler automatically by a HW/SW codesign system called C-DASH [1] [2]. C-DASH is an ISA based processor description language, and is used in this paper to presents an indirect approach for generating a compiled simulator and the automatic generation of a simulation compiler.

The rest of this paper is organized as follows. Section 2 presents related works addressing instruction set simulation techniques. The indirect simulation compiler we propose is described in section 3. The technique for automatic generation of simulation compiler by C-DASH is presented in section 4. As an evaluation of our approach, the simulation time is shown in section 5. Section 6 concludes this paper.

2. Related Works

This section provides an outline of related works on

ISA simulators; which can be classified into two types: 1) Interpretive simulation and 2) Compiled simulation. An interpretive simulator is very slow; so to resolve this problem, some solutions are proposed. A hybrid interpretive/compiled scheme is described in [3]. In this system, self-modifying codes are interpretively executed. In [4], a virtual machine approach is described in which instructions for a target code are translated into the instructions for the virtual machine. Subsequently, the instructions for the virtual machine are translated into instructions for a host machine. Embra [5] and Shade [6] simulators use dynamic binary translation and result caching to improve simulation performance. Embra is not retargetable, and is restricted to the simulation of MIPS R3000/R4000 architecture. Shade has several versions such as Shade-V8.V8, -Mips.V8 and -V9.V8; indicating that the target machines are V8 SPARC, V9 SPARC, MIPS and the host machine is V8 SPARC. In [7] and [8], a JIT-CCS (just in time cache compiled simulation) technique is proposed. This technique integrates the simulation compiler into the simulator. The compilation of an instruction takes place at the simulator run-time, just-in-time before the instruction is going to be executed. Subsequently, the extracted information is stored in a simulation cache for direct reuse in a repeated execution of the program address. The simulator recognizes if the program code of a previously executed address has changed and initiates a re-compilation. This technique makes an assumption to get performance closer to the compiled simulation: the number of repeatedly executed instructions should be very large, such that 90% of the execution time is spent on just 10% of the code. However, this assumption may not hold true for all real world applications [7].

Retargetable fast simulators based on an ADL (Architecture Description Language) have been proposed within the framework of nML [9], ISDL [19], MIMOLA [10][11] LISA [12][13], EXPRESSION [14][15][16] and ASIP Meister [17][18].

To summarize what we have discussed so far. Many ISA simulators are interpretive. To improve the simulation speed, some simulators adopt compiled simulation. With compiled simulation, the code of a target machine is directly translated into the code of a host machine (binary to binary). However, the binary-to-binary translation approach depends on the host machine and is not portable.

This paper focuses on automatic generation of a simulation compiler that is flexible, portable and a reasonably high performance ISA simulator. We propose an indirect approach using a compiler on the host machine. In this approach, to improve the performance of the simulation, time consuming decoding is moved into compile time. The indirect approach makes the structure of a simulation compiler simple and improves

Figure 1. Structure of ISA Simulator

portability because the indirect approach is independent of any host machine.

3. ISA Simulator

ISA simulators can be classified into 2 types. One is interpretive simulation and the other is compiled simulation. An interpretive simulation is very flexible but performance is very slow. To improve simulation performance, the execution of interpretive simulation is divided into two phases (Compile time and Run time) by the simulation compiler. Figure 1(1) and (2) shows the structure of interpretive and compiled simulators, respectively. Interpretive and compiled simulations are tradeoffs between flexibility and performance on applications.

3.1. Interpretive Simulator

An interpretive simulator is basically a virtual machine implemented in software. Interpretive simulators process instructions using a software model of the target processor. Interpretive simulators can be summarized as a loop in which instructions are fetched, decoded and executed. The performances of interpretive simulators are very slow. Especially, unlike in hardware execution, the decode stage requests many clocks in software execution. Advantages of this model include ease of implementation, flexibility and the ability to garner varied processor state information.

3.2. Compiled simulator

To improve simulation performance, a compiled simulation approach is proposed. This approach translates the target instructions into host instructions. The translation process can be organized in two ways (direct or indirect approaches).

The direct approach directly translates target machine binary to host machine binary, its advantage is fast translation and simulation times, but the translator is complex and less portable because instructions depend on the host machine, and a translator is necessary for each host machine.

Figure 2. Compiled Simulator

```
add %r1, %r2, %r3
```

Figure 3. Code of Target Machine

```
ld [%vs + vs_r1], %s1
ld [%vs + vs_r2], %s2
add %s1, %s2, %s3
st %s3, [%vs + vs_r3]
inc 4, %vpc
```

Figure 4. Code of Host Machine

The indirect approach divides the translation process into two parts. 1) Compile the target code to a program written in a high level language such as C or Java. 2) Compile the program into host code with a compiler on the host machine. This approach takes advantages of the existing compilers on the host machine, simplifies the implementation of a translator and improves portability. But simulation performance is slower than that with the direct approach.

Figure 2 shows the structure of the translation process in direct approach. The translator includes a TLT (Translation Lookup Table) that has a table for corresponding between a target code and a host code. Target code is translated into host code based on the TLT; Figure 3 and Figure 4 are sample instruction codes of the target and host, respectively, that simulate it. In Figure 3, %rN means general purpose register of the target machine. The sample program executes add instructions (add the %r1 and %r2 then write back the result into the %r3). The basic idea of a compiled simulation is to create parameters that indicate a virtual register such as vs_rN and vpc in Figure 4. vs_rN and vpc represent a general purpose register and the program counter of the target machine, respectively. %sN indicates the registers of the host machine.

3.3. Our Approach

In our approach, target code is translated into a high level language by the simulation compiler. The translation process resembles the structure of a disassembler. The advantages of our approach are reasonable execution speed, flexibility and portability.

Figure 5. Flow of Our Approach

Similar to other simulation compilers, our approach moves the time consuming instruction decoding process into the compile time phase to improve performance. Our approach can get reasonable performance while maintaining the flexibility of interpretive simulation. Figure 5 shows the translation flow of our approach. 1) Compile App (Application) into Obj (Object file) for target processor. 2) SC (Simulation Compiler) translates Obj into MF (Modeling File) in the C language. 3) MF and IDF (Instruction Definition File) are compiled into EXE (execution file) on a host machine 4) Execute simulator.

The SC has an object code analyzer that analyzes the target code of an application program, and obtains the operation code and address information of the object code. The SC generates the label name as the address information in a high level language. The SC translates operation code into a mnemonic name, and decides operands and generates a description that represents the operand in a high level language.

The flows are implemented in the C language. The structure of the SC is shown in Figure 6. The SC repeats fetch() and decode() until the end of operation code. The fetch function sets an instruction word into "ir" representing the instruction register and increments "pc", representing the program counter, by N bytes, representing the word size. The decode function decodes one binary instruction at a time to generate a modeling file for the input application. Figure 7 shows the binary and assembly of the instruction.

The IDF consists of instruction definitions such as in Figure 8. The MF is shown in Figure 9. The IDF and MF are formalized simulators. Each instruction of a processor is defined in C, and the object code

```

main() {
    while(!end_of_operation_code) {
        fetch(pc);
        decode(ir);
    }
}

fetch() {
    ir = mem[pc];
    pc=pc+N;
}

decode() {
    if((ir&ADD_MASK)==ADD_OP) {
        printf("L%06d : add (" ,pc-N);
        getRegisterName((ir&DST)>>11);
        getRegisterName((ir&SRC1)>>16);
        getRegisterName((ir&SRC2)>>21);
        printf(");");
    }
    ...
}

getRegisterName(int addr) {    // decide the register name
    switch(addr) {
        0 : printf("%r0"); return;
        1 : printf("%r1"); return;
        ...
    }
}

operandIMM() {}
operandMEM() {}

```

Figure 6. Implementation of SC

Size	6	5	5	5	5	6
Field	opcode	Rs	Rt	Rd	0x0	func
Binary	000000	00001	00010	00011	00000	100000
Mask	111111	00000	00000	00000	00000	111111
DST	000000	00000	00000	11111	00000	000000
SRC1	000000	11111	00000	00000	00000	000000
SRC2	000000	00000	11111	00000	00000	000000
Assembly	add	Rd	, Rs	, Rt		

Figure 7. Binary and Assembly

of the target processor is translated into a MF in C by the SC.

The performance of the simulator depends on the number of features it has. Therefore, the simulator generated by the simulation compiler in the C language includes `#ifdef` and `#endif` compiler directives in order to make the performance of the simulator's reasonable speed correspond to the required features. Figure 8 shows a description of an add instruction for a target processor. This instruction has 3 arguments (2 source register and 1 destination address). In Figure 8, `src1` and `src2` represent the data of a source register, and `dst` is a pointer to the destination register. The `set_flag()` function checks the data of the execution result and sets flags such as `zero_flag`, `minus_flags` etc. Figure 9 shows the modeling of the target code shown in Figure 3. The format of Figure 9 is `<label_name> mnemonic (arguments)`. In the arguments, `reg_f[1]`, `reg_f[2]` and `reg_f[3]` indicate general purpose registers and represent `%r1`, `%r2` and `%r3`, respectively, in Figure 3.

Figure 8 and Figure 9 are translated in Figure 10 and Figure 11, respectively, by a compiler on the host

```

add_op(int *dst, int src1, int src2) {
    dst = src1 + src2;
    set_flag(dst,src1,src2);
#ifdef INTERRUPT
    check_interrupt();
#endif
#ifdef PROFILE
    set_profile();
#endif
}

```

Figure 8. Instruction of Target

```

L00100000  add(reg_f[1], reg_f[2], reg_f[3]);
....

```

Figure 9. Modeling of the Target Code

```

_add:
    pushl    %ebp
    movl     %esp, %ebp
    movl     16(%ebp), %eax
    movl     12(%ebp), %edx
    addl     8(%ebp), %edx
    movl     %edx, (%eax)
    incl     _pc
    popl     %ebp
    ret

```

Figure 10. Representation of add function

```

L16:
    movl     $1, (%esp)
    movl     $2, 4(%esp)
    movl     $_reg_f+4, 8(%esp)
    call     _add

```

Figure 11. Representation of function call

machine.

In our approach, if self-modifying codes exist, these behaviors are described in each instruction by defining the corresponding behavior to the bit pattern.

SC and IDF are necessary for a target processor in order to automatically generate a compiled simulator. An ISA based HW/SW co-design system easily generates the SC and IDF. We use C-DASH to generate this required information.

4. Outline of C-DASH

C-DASH adopts an ISA based processor description in CHDL (C-DASH HDL), which is an original language to describe processors. A CHDL description is composed of resource declarations and instruction descriptions for a target processor. C-DASH can generate HW (synthesizable HDL descriptions of a processor) and SWDE (such as a compiler, a simulator, an assembler and a disassembler) for the processor by extracting necessary information from a processor description in

```

module MODULE_NAME
  RESOURCE_DECLARATION
  INSTRUCTION_DEFINITION
endmodule

```

Figure 12. Structure of Module

```

reg      REG_NAME BIT_WIDTH [ SIZE ];
ram      RAM_NAME BIT_WIDTH SIZE;
instruct INST_MEM  PC  IR;

```

Figure 13. Resource Declaration

CHDL. C-DASH allows a processor designer to reduce the effort involved in constructing HW and SWDE. Since the processor description in CHDL is based on ISA, instructions can be changed flexibly and easily. C-DASH can develop RISC, CISC and stack architecture based on a single clock, and a pipelined processor.

4.1. Method of Processor Definition

To define a processor in CHDL, both resource declarations and instruction definitions are required. Figure 12 shows the structure. The "module" consists of MODULE_NAME, RESOURCE_DECLARATION and INSTRUCTION_DEFINITION. MODULE_NAME indicates the processor name, RESOURCE_DECLARATION indicates resource declarations and INSTRUCTION_DEFINITION indicates an instruction set description.

In the RESOURCE_DECLARATION, general purpose registers, specific registers (such as program counter, instruction register) and memory declarations with bit-width and data-size, are required.

Figure 13 shows the structure of resource declarations. The "reg" keyword is used to declare the register name (REG_NAME) and bit width (BIT_WIDTH). The "ram" keyword is used to declare the memory name (RAM_NAME) and bit width (BIT_WIDTH) and address size (SIZE). To indicate an instruction memory (INST_MEM), a program counter (PC) and an instruction register (IR), an "instructin" keyword is used.

To define an instruction set, instruction descriptions that consist of simulator and assembler descriptions are required. There are two methods for describing instructions. 1) Sequential description is a very simple description based on instruction behavior. The architecture of a processor consists of four fixed stages (Fetch, Decode, Execute and Write back). 2) Pipeline description is based on the behavior of each stage and can change the architecture of a processor. Figure 14 shows an instruction definition using an "instruct" keyword. The "instruct" consists of a mnemonic name (MNEMONIC), a bit pattern corresponding to the mnemonic (BIT_PATTERN), and the behavior of the instruction (BEHAVIOR). The behavior descriptions, which include the operation of memory access, are

```

instruct MNEMONIC { BIT_PATTERN } { BEHAVIOR }

```

Figure 14. Instruction Definition (sequential)

```

fsm { STAGE_NAME ':' '{' BEHAVIOR '}' }
instruct MNEMONIC { BIT_PATTERN } {
  STAGE_NAME ':' '{' BEHAVIOR '}'
}

```

Figure 15. Instruction Definition (pipeline)

```

asm MNEMONIC { FORMAT } { OPERAND }

```

Figure 16. Assembler Definition

```

instruct add {000000 sssss tttt dddd 00000 100000} {
  reg_f[d] = reg_f[s] + reg_f[t];
}
asm add {<label> <opcode> <reg>, <reg>, <reg>} {
  d = number($3);
  s = number($5);
  t = number($7);
}

```

Figure 17. Example of Add Instruction in

assumed to be finished in a single clock. Figure 15 shows a description of a pipeline. To define instructions of a pipeline processor, a definition staged by using a "fsm" keyword, and an instruction definition by using an "instruct" keyword are required. Common operations (such as fetch and decode) to all instructions are described in BEHAVIOR of "fsm". Operations depend on how the instruction is described in the BEHAVIOR of the "instruct".

To define an assembly language format, an "asm" keyword is used. The "asm" consists of the mnemonic name (MNEMONIC), an instruction format (FORMAT) and an operand description (OPERAND). FORMAT indicates the structure of the operand for MNEMONIC. OPERAND represents the correspondence of a sub bit of the BIT_PATTERN and the register address and/or immediate value. Figure 16 shows an assembler description by using the "asm" keyword. An example of an add instruction described in Figure 3, is described in CHDL as follows.

The "s", "t", "d" represent a 5 bit width sub-bit in the bit pattern and indicate the address of register files that represented "reg_f[]". The "number(\$N)" function returns the address of the register file such as %r1, %r2 and %r3, as 1, 2 and 3, respectively. The \$N is the number of an argument, which includes ",", in assembly language.

4.2. Generation of an Intermediate Representation

C-DASH analyzes the instruction definition in CHDL and generates an IRPD (intermediate representation of processor definition) in CHDL. Subsequently C-DASH generates HW and SWDE based on an IRPD.

INSTRUCTION_N {		
id	{ ID }	
mnemonic	{ MNEMONIC }	// add
bit_pattern	{ BIT_PATTERN }	// 000000 sssss ttttt ...
mask	{ MASK }	// 111111 00000
opcode	{ OPCODE }	// 000000 00000100000
sub_bit	{ SUB_BIT }	// {s=ir[25:21],\$5,REG},...
behavior	{ BEHAVIOR }	// dst = src1 + src2
format	{ FORMAT }	// <opcode> <reg> , <reg> , <reg>
operand	{ OPERAND }	// dst=reg_f[d], src1=reg_f[s]...

Figure 18. Structure of IRPD

```

decode () {
    if ((ir & MASK_N) == OPCODE_N) { // decode
        FORMAT_N; //
    }
}

```

Figure 19. Structure of a Simulation Compiler

"MASK" is used to determine which instruction the bit-pattern is. If the bit-pattern is 1 or 0, the mask pattern corresponding to the bit pattern is 1, otherwise 0. The "SUB_BIT" represents the operand type (such as register REG, immediate IMM), bit width and position corresponding to "ir" register.

4.3. Generation of a Simulation Compiler

The structure of a SC resembles the structures of a disassembler. As shown in Figure 6, the main part of the SC is the decode() function. In order to automatically generate a SC, the structure of decode() is summarized such as in Figure 19. MNEMONIC_N, OPCODE_N and FORMAT_N mean MNEMONIC, OPCODE and FORMAT of INSTRUCTION_N respectively. <opcode> in FORMAT is replaced by MNEMONIC and <reg>s in FORMAT are replaced by the getName() function which obtains the register name. It is possible to generate a SC by outputting necessary information from IRPD.

To generate an IDF such as in Figure 8, MNEMONIC and BEHAVIOR are used and additional information, which is accompanied by the compiler directive, is added automatically.

5. Evaluation

To evaluate our approach, we implement simple applications. One is a sort program, and another is a FIR (Finite Impulse Response) filter program.

The results showed that indirect compilation simulation is slower than direct execution of the application, but faster than interpretive simulation.

Simulation time reduction is an important issue. Unlike hardware design, the slowest part is the decoder, because comparisons are performed many times for each instruction using many if-else or switch-case statements.

Table 1. Result of Execution

	Compile-Time(sec)	Run-Time(sec)	Total (sec)
Host	~0	2	2
Compiled	3	10	13
Interpreter	0	80	80

6. Conclusion

In this paper, we focused on an ISA simulator for rapid prototyping. The construction of an ISA simulator depends on its proposed use. To rapidly develop a processor, a faster ISA simulator is required. A compiled simulation that translates target binary to host binary is complex and has less flexibility. In our approach, target codes are translated into a high level language and then translated into host code with a compiler. Our approach makes the translator simple and portable, because it takes advantage of the compiler on the host machine.

In the future, we would like to evaluate an actual application that is more complex. We also intend to create an algorithm of re-compilation for the simulator because large programs are compiled many times with very little change.

References

- [1] Hideaki Yanagisawa, Minoru Uehara, Hideki Mori, "ISA based system design language in HW/SW co-design environment", In Proc. of 13th IEEE International Workshop on Rapid System Prototyping 2002, pp.122-127, 2002.
- [2] Hideaki Yanagisawa, Minoru Uehara, Hideki Mori, "Development of the Design Automation System for ASIP based on ISA", In Proc. of International Conference on Computer, Communication and Control Technologies (CCCT'03), Vol. 3, pp.271-276, 2003.
- [3] Vojin Živojnovic, Heinrich Meyr, "Compiled HW/SW co-simulation", In Proc. of the 33rd annual conference on Design automation conference, pp. 690 - 695, Las Vegas, Nevada, United States, June 1996.
- [4] J. Zhu, D. Gajski, "A Retargetable, Ultra-Fast instruction Set Simulator", In Proc. of Design, Automation and Test in Europe (DATE '99), pp. 298-, March 1999.
- [5] Emmett Witchel, Mendel Rosenblum, "Embra: fast and flexible machine simulation", In Proc. of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pp. 68 - 79, Philadelphia, Pennsylvania, United States, May, 1996
- [6] Bob Cmelik, David Keppel, "Shade: a fast instruction-set simulator for execution profiling", In Proc. of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems, pp.128 - 137, Nashville, Tennessee, United States, May, 1994.
- [7] Mehrdad Reshadi, Prabhat Mishra, Nikil Dutt, Instruction set compiled simulation: a technique for fast and flexible instruction set simulation, In Proc. of the 40th

- conference on Design automation (DAC'03), pp. 758 - 763, Anaheim, CA, USA, 2003.
- [8] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, Andreas Hoffmann, "A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation", In Proc. of the 39th conference on Design automation (DAC'02), pp. 22 - 27, New Orleans, Louisiana, USA, June 10 - 14, 2002.
 - [9] A. Fauth, J. Van Praet, M. Freericks, "Describing Instruction Set Processors Using nML", IEEE, European Design and Test Conf., 1995.
 - [10] R. Leupers and P. Marwedel. "Retargetable code generation based on structural processor descriptions", Design Automation for Embedded Systems, 1998.
 - [11] Steven Bashford, Ulrich Bieker, Berthold Harking, Rainer Leupers, Peter Marwedel, Andreas Neumann, Dietmar Voggenauer, "The MIMOLA Language Version 4.1", University of Dortmund, September 1994.
 - [12] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen and H. Meyr: "A Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using the Machine Description Language LISA." International Conference on Computer Aided Design (ICCAD), November 2001.
 - [13] A. Hoffmann, A. Nohl, S. Pees, G. Braun, and H. Meyr, "Generating Production Quality Software Development Tools Using a Machine Description Language, "International Conference on Design Automation and Test in Europe Conference (DATE), 2001.
 - [14] Ashok Halamebi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Duttand, Alex Nicolau, "EXPRESSION : A Language for Architecture Exploration through Compiler/Simulator Retargetability", in Proc. Design Automation and Test in Europe (DATE 99), 1999.
 - [15] Asheesh Khare, Nick Savoiu, Ashok Halambi, Peter Grun, Nikil Dutt and Alex Nicolau., "VSAT: A Visual Specification and Analysis Tool for System-On-Chip Exploration", EUROMICRO 99,
 - [16] Peter Grun, Ashok Halambi, Ashesh Khare, Nikil Dutt, Alexandru Nicolau, "EXPRESSION : An ADL for System Level Design Exploration", Department of Information and Computer Science, University of California, Irvine, Technical Report #98-29, September 1998.
 - [17] Makiko Itoh, Yoshinori Takeuchi, Masaharu Imai, Aki-chika Shiomi, "Synthesizable HDL Generation for Pipelined Processors from a Micro-Operation Description", IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E83-A, No. 3, pp.394-400, March 2000.
 - [18] Shinsuke Kobayashi, Kentaro Mita, Yoshinori Takeuchi, and Masaharu Imai, "A Compiler Generation Method for HW/SW Codesign Based on Configurable Processors", IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E85-A, No. 12, pp.2586-2595, Dec. 2002.
 - [19] George Hadjiyiannis, Pietro Russo, Srinivas Devadas, "A methodology for accurate performance evaluation in architecture exploration", pp. 915-920, In Proc. of the 36th ACM/IEEE conference on Design automation conference (DAC'99), 1999.