A Compiled-Code Simulation Technique for RTL Designs

Yi-Jung Yeh, Kai-hui Chang, Ming-tang Chen, Sy-yen Kuo

Department of Electrical Engineering, National Taiwan University

ABSTRACT

A compiled-code technique for register transfer level (RTL) simulation is presented. It converts the evaluations to C code while keeping the interpretive scheduler. The compiled-code technique can accelerate the simulation speed, which saves up to about 79% of the simulation time, especially in the designs with complex evaluations.

1. Introduction

As the technology of VLSI improves, logic simulation is getting more and more important, and various techniques that increase the speed of simulation are developed. Those techniques can be classified into two categories [1]. In interpretive simulation, a central scheduler traverses the data structures and calls the evaluation routines iteratively. In compiled-code simulation, the fact that the design will not change during simulation is used and can produce customized program that runs efficiently. Previous works on compiled-code simulation include compiled-code event-driven logic simulator [2], Shadow algorithm [3], Inversion algorithm [4], [5], and Branching Programs [6]. All of them focused on gate-level simulation. In the current design strategy, register transfer level (RTL) descriptions are used to verify the correctness of the algorithm or to be synthesized into gate-level circuits. So the RTL simulation is also very important. There are some commercial compiled-code simulators like NC-Verilog from Cadence [7] and VCS from Synopsys. Their approach is to write a new simulator which generates machine code directly. The advantage is that the performance can be optimized to a maximal. The disadvantage is that it takes a very long time to develop a totally new simulator, and takes more effort to maintain two different simulators. In this paper, we proposed an RTL compiled-code technique, which can accelerate the simulation speed of RTL designs as well as gate-level designs.

The simulator described in this paper is called VCK (Verilog-C Kernel) [8] and uses Verilog hardware description language as its source language. Verilog is a complete and complex language which can be used to design and simulate the hardware. But no matter how complex the hardware description language is designed, the idea is very simple: The state of the circuit is stored in some variables. There are assignments between these variables. There are also some events waiting to trigger these assignments. What the simulator needs to do is to trigger the assignments at the right time and update the variables according to the evaluation results of the assignments.

In VCK, the source file will be converted into many blocks of statements called “states”. There are assignments in the states. These assignments are grouped in a state because they share the same triggering conditions. There is a scheduler determining the state to be executed at a time point. The author will focus on the assignments in a state and discuss the compile-code techniques that applied to it.

2. Compiled-code Techniques

2.1 Basic Concepts

The reason why compiled-code technique is beneficial is from the fact that the design will not be changed during simulation; therefore, specific evaluation routines can be used instead of generic ones. So there are two different parts of code to be generated. One is pre-compiled specific evaluation routines, and the other one is the design-specific C state routines that call those evaluation routines.

2.2 Evaluation Routines

The evaluation routines are simplified from generic ones. When simplifying the routines, computer architecture must be taken into account. The computers being used now are mostly 32-bit; so 32-bit is a boundary to our simplification. All operations less than 32 bits are all the same in current computers. For example, 1-bit bit-or is the same complex as 2-bit bit-or. 32-bit itself is also a special case. For example, no mask is necessary if the operands are all 32 bits. 64 bits is also a special case and is treated separately. All bit-length above 32 bits and is not 64 bits are evaluated using the generic routines.

2.3 Data Structures and Their Expansions

In VCK, the design is converted to a list of states which consist of one or more assignments. Each assignment is composed of variables and equations. These data structures are discussed below. The method to convert them into C code, which is called “expansion” in this paper, is also discussed.

2.3.1 Variables

Variable is the basic data storage in VCK. It stores the simulation value and some other information of this variable. The simulation value is stored in a data type called “value_t”, which can be used to store the Verilog 4-value data. Each variable has a unique ID called “variable_id”, which will be used in the C code generated.

2.3.2 Equations

There are two basic constructs in the RHS of assignments and in conditions: The variables (or operands)

and the operations. Variable has already been discussed. Equation is a data structure that connects the variables with the operations that manipulate them. Every equation has a value_t called "sim_v" which keeps the current simulation value of the equation.

There are three types of equations: unary, binary and associative. Unary equations are operations that only have one operand like "negation". Binary equations are operations that have two operands like "division". Associative equations are operations that are associative like "addition". Each equation has a field called "operation_type" that represents the operation of the equation. Equation also has fields that point to variables or other equations. Figure 1 shows some examples and illustrate the usage of them. The "operation_type" of the equation is enclosed in the parentheses. The variables are written directly using their original variable names. There are two pointers used in an equation to connect to other equations or variables, which are named "side" and "down", and is represented in the graph by drawing "side to" or "down to" the equation. The expansions of these data structures are also given in the figure. Routines "value_negate", etc. are evaluation routines corresponding to the "operation_type". Variable "ret" is the result of the evaluations.

Type	Example	Data Structure	Expansion
Unary	!a	equation(!) | a	value_negate(a, ret);
Binary	a / b	equation(/) | a – b	value_divide(a, b, ret);

Associa-tive	a + b + c	equation(+)
		|
		a – b – c
			value_add(a, b, ret);
			value_add(ret, c, ret);
			……

Figure 1. Equation examples

2.3.3 Simple Assignments

Simple assignment is an assignment without any condition. The data structure is given in Figure 2. There is a pointer to the variable in the LHS of the assignment, and a pointer to an equation or variable representing the RHS of the assignment. There is also some other information about the assignment, like bit-select or part-select, which selects only a portion of the bits to be assigned.

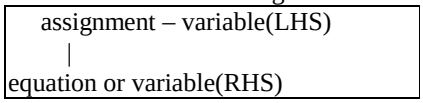

Figure 2. Simple Assignment

To expand a simple assignment, a copy from the RHS to the LHS is enough, like:

```
value_copy(RHS, LHS);
```

If there is a part-select or bit-select in the assignment, another routine is used:

```
part_copy(RHS, msb2, lsb2, LHS, msb1, lsb1);
```

If the RHS of the assignment is an equation, it is first expanded before the assignment expansion.

2.3.4 Conditional Assignments

Conditional assignment is an assignment with a condition. To execute a conditional assignment, the condition is first evaluated, and then the result is used to select a block of assignments to execute according to the tag of the block. The data structure is given in Figure 3.

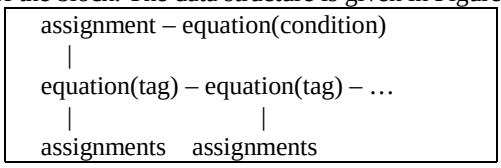

Figure 3. Conditional Assignment

There are two kinds of conditional assignments. The first one is simple conditional assignment, and the second one is complex conditional assignment. In simple conditional assignment, the condition is simple and can be represented by a jump-table. That is, all cases that the condition can generate can be enumerated. Efficient C code using "switch" can thus be generated. For example, in the following Verilog case structure:

```
case (a)
        1'b0: b = 0;
        1'b1: b = 1;
        default b = 'bx;
endcase
```

Assume the variable_id of "a" is 0, the variable_id of "b" is 1, the variable_id for "0" is 2, the variable_id for "1" is 3, and the variable_id for "x" is 100. (Constant number is also treated as a variable in VCK.) Then the following C code will be generated:

```
{
  int index = value_to_int(value0);
  switch (index)
  {
  case 0:
      value_copy(value2, value1);
      break;
  case 1:
      value_copy(value3, value1);
      break;
  case 2:
      value_copy(value100, value1);
      break;
  default:
      break;
  }
}
```

Where "value_to_int" converts the Verilog 4-value data in value0 to an integer. If the data in value0 is "X" or "Z", then index will be 2. The case conditions in C

correspond to the tags in the data structure of conditional assignment. The statements under "case" correspond to the assignments that are attached to the tags, which use "value_copy" to copy the first operand to the second.

In complex conditional assignments, all the cases that the condition can generate cannot be enumerated, so more complex if-then structure must be used to generate the C code. Each tag becomes an "if" statement, and the assignments under it become the "then" part. The end of the "then" part is a statement which jumps out of the conditional assignment.

For example, a statement like:

```
if (a < b)
      c = d;
```

Will be converted to something like:

```
if (value_compare_case_lt(value0, value1))
{
      value_copy(value2, value3);
      goto label0;
}
label0:
next statement……
```

The variable_id of "a" is 0, the variable_id of "b" is 1, the variable_id of "c" is 3, and the variable_id of "d" is 2. Routine "value_compare_case_lt" compares two values and returns true if the first operand is less than the second.

2.3.5 Assignment Expansion Algorithm

Procedure "assignment_generate" is the main routine that generates the code of an assignment. The procedure "generage_sa" is for simple assignment, while procedure "generate_ca" expands conditional assignments.

```
procedure assignment_generate()
    foreach (assignment A in the state) do
        if (A is a simple assignment) then
            generate_sa(A);
        else
            generate_ca(A);
end

procedure generate_sa(assignment A)
    R = equation_expand(RHS of A);
    generate_code(assign R to LHS);
end

procedure generate_ca(assignment A)
    C = equation_expand(condition of A);
    foreach(tag T of A)
        generate_code(compare C with T);
        assignment_generate(assignments under
T);
        generate_code(goto end_label);
    generate_code(end_label);
end
```

There are some differences between complex conditional assignment and simple conditional assignment. In simple conditional assignment, the line "compare *C* with *T*" generates a "case" statement. While in complex conditional assignment, an "if-then" statement is generated.

3. Interpretive to Compiled-Code Conversion

In compiled-code VCK, states are expanded to C routines called "state routine", and assignments inside them are expanded to C code. Since the scheduler is still interpretive, there are two problems to be solved. The first one is how to call correct C state routines from the scheduler. And the second one is that the C code generated must be able to use the same data structures as those at compilation time because the scheduler depends on them.

3.1 State Routine Calls

To call the state routines, an array of pointers to the state routines is built at the compilation time. This way any state routine can be called using an index, which is its state number. A compiled code example is shown below, where the global array state_ctask_g[] keeps the pointers to the routines:

```
int (*state_ctask_g[])() =
{
    state_1_ctask,
    state_2_ctask,
    state_3_ctask
};
int state_1_ctask(){…}
int state_2_ctask(){…}
int state_3_ctask(){…}
```

Each routine "state_n_ctask()" represents a state. In the scheduler, it calls the correct routine using the following code with state number n.

```
int (*addr_ctask)();
addr_ctask = state_ctask_g[n];
result = (*addr_ctask)();
```

Where "addr_ctask" is a variable which stores the address to the current routine obtained from state_ctask_g, and "result" is an integer for the return value of the state routine.

3.2 Data Structure Conversions

In order to recover back the same data structures as those at compilation time, we write out the data structures at the compilation time and read them back at the simulation time. The data structures used in the compiled-code part are mostly variables and equations. There are two kinds of information saved in the data structures: static and dynamic. Static information includes the bit-length of the variable and the current simulation value, etc. This information will not be changed between the compilation time and the simulation time. Dynamic information includes pointers to other data structures, which could be different between the compilation time and the simulation time. To write out the static information, just save them into a file. To write out dynamic information, we give each variable and equation a unique identifier, which is an integer. The identifier of the variable is called variable_id, and the identifier of the

equation is called equation_id. When writing out pointers to a file, write the identifier numbers instead of the values of the pointers. When the design is reloaded at simulation time, the memory of the data structures are reallocated, and a table which maps the identifiers to the addresses of the data structures is built. Later in simulation, the compiled-code part can get the addresses of the pointers by this mapping table. The conversion process is shown in Figure 4.

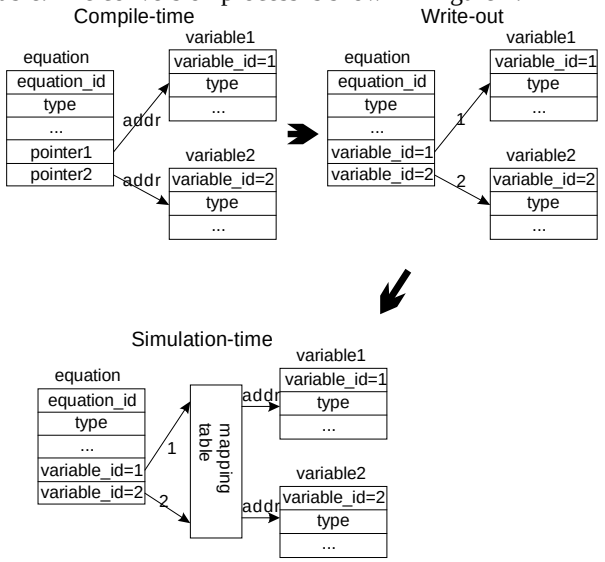

Figure 4. Pointer Conversions

4. Experimental Results

There are four test cases used in the experiments. Mul16 is a Wallace Tree 16*16 multiplier. MC6805 is a CPU core. AAL1 and AAL2 are ATM chip designs. Mul16 is gate-level. MC6805, AAL1 and AAL2 are RTL.

We conducted all the experiments on two platforms. PC is Intel Pentium III 733MHz with 256M memory. SUN is Sun UltraSparc 440MHz with 256M memory. Preprocessing time includes compiled-code generation and compilation. The results are shown in Table 1.

Table 1. Experimental Results

Test \ Method	Interpretive (sec)	Compiled-code(sec)		Time saved (%)
		Preprocessing	Simulation	
Mul16(PC)	444.8	7.2	94.0	79
Mul16(SUN)	1212.4	7.9	274.5	77
MC6805(PC)	35	15.9	26.3	25
MC6805(SUN)	89.4	32.9	58.8	34
AAL1(PC)	90.3	4.1	55.8	38
AAL1(SUN)	326.5	8.7	194.8	40
AAL2(PC)	1251.8	9.0	665.3	47
AAL2(SUN)	3968.1	14.5	2108.3	47

5. Conclusions

A register transfer level compiled-code simulator is proposed in this paper. An approach and the algorithm to convert data structures from interpretive simulation to compiled-code are proposed. Many optimization issues are also discussed. From the experimental results in Table 1, it can be seen that the acceleration saves simulation time between 25% to 79%, depending on the design. An average acceleration of about 48% saving in simulation time is achieved from the technique proposed in this paper. In general, designs with more complicated evaluation are more beneficial to use the compiled-code technique.

Compile time is also an important factor to judge the performance of compiled-code simulator. Larger circuits usually require longer time to compile. However, larger C code also means greater optimization. If the running time is long enough, the time used to compile the design can usually be compensated by the time saved in simulation.

6. References

[1] Z. Barzilai, J. L. Carter, and J. D. Rutledge, “HSS – A high-speed simulator.,” IEEE Transactions on Computer-Aided Design, vol. 6, July 1987, pp. 601-617.

[2] D. M. Lewis, “A hierarchical compiled-code event-driven logic simulator,” IEEE Transactions on Computer-Aided Design, June 1991, pp. 726-737.

[3] Peter M. Maurer, “The Shadow algorithm: A scheduling technique for both compiled and interpreted simulation,” IEEE Transactions on Computer-Aided Design, vol. 12, September 1993, pp. 1411-1413.

[4] Peter M. Maurer, “The Inversion algorithm for digital simulation,” IEEE Transactions on CAD of Intergrated Circuits and Systems, Vol. 16, NO. 7, July 1997, pp. 762-769.

[5] P. Maurer and W. Schilp, “Three-valued simulation with the inversion algorithm,” Department of Computer Science Engineering., University of South Florida, Tampa, Tech. Rep. DA-27, 1995.

[6] Pranav Ashar and Sharad Malik, “Fast Functional Simulation Using Branching Programs,” IEEE 1995, pp. 408-412.

[7] NC-Verilog Help, Version 1.1, Cadence Design Systems, 1997.

[8] VCK User’s Manual, Version 1.1, Avery Design Systems, 2001.