A Hybrid Logic Simulator Using LUT Cascade Emulators

Hiroki Nakahara Tsutomu Sasao Munehiro Matsuura

Department of Computer Science and Electronics, Kyushu Institute of Technology,
680-4, Kawazu, Iizuka, Fukuoka, 820-8502, Japan

Abstract— This paper presents a hybrid logic simulator using both an event-driven and a cycle-based methods. For special primitives such as memories and tri-state buffers, it uses an event-driven method. For other parts, it uses a cycle-based method using LUT cascade emulators. To simulate a large scale circuit, it partitions the circuit into smaller ones, and realizes each part by an LUT cascade emulator. Next, it combines these emulators by interconnections. Since a multiplier often requires large memories in an LUT cascade, an instruction of the processor is used instead of the LUT cascade. This will reduce the code size and the simulation time. Our experiment shows that proposed method is effective for circuits including arithmetic operations.

I. Introduction

With the increase of the integration of LSIs, the time for the verification of the design increases. Thus, a high-speed logic simulators is needed. We have proposed a new type of cycle-based logic simulator[10, 8, 9]. In this simulator, first, the given circuit is converted into LUT cascade, then the cell data is stored into the memory on the PC. Also, C codes for the control circuit that controls the LUT cascade emulator is generated. The logic simulation is executed on a virtual LUT cascade emulator implemented on the memory of the PC. Although our simulator using a standard PC is slower than a hardware-based emulator, it is much cheaper. Also, the performance can be enhanced with the improvement of the PCs. Proposed method is based on a table-lookup method, which uses large and cheap memory on a PC. Our simulator outperforms the Levelized Compiled Code (LCC) [1], BDD-based logic simulator [5], and commercial logic simulators [3, 6]. The tricks of our fast simulation are:

1 Multi-input and multi-output cells to simulate many gates. These reduce the number of memory references.

2 Code that utilizes both instruction cache and data cache on the processor.

3 The memory packing [11] that reduces the cache misses.

In [10, 8], to simulate a large scale logic circuit, outputs of the circuit are partitioned. However, when a component function is excessively complex, this method fails. In [9], we presented a method to partition the netlist. Although this method requires an extra time for evaluating connections between partitioned networks, it is applicable to a large scale circuit.

In the previous method, special primitives, such as tri-state buffer and memory, could not be simulated. To simulate such primitives and to reduce the simulation time, in this paper, we use a hybrid logic simulator using LUT cascade emulators. When arithmetic circuits, such as multiplier, are represented by the LUT cascade, the necessary size of memory will be quite large. Also, the special primitives such as memories and tri-state buffers cannot be represented by the LUT cascade. Thus, we simulate such special primitives by an event-driven method, and simulate arithmetic circuits by instructions of the processor [1] rather than the LUT cascade. For the other parts, we use the LUT cascade emulator: we generate the cell data for the LUT cascade and the codes for the control part of the LUT cascade emulator. To generate execution code, all generated codes are linked. Finally the given circuit is simulated by the execution code using the cell data. The proposed method has the following advantages:

1. It can simulate special primitives (e.g. memories and tri-state buffers) those could not be treated in the previous method.

2. It is faster than the previous method, since the arithmetic primitives (e.g. multiplier and divider) are directly simulated with instructions of the processor.

To confirm the effectiveness of the proposed method, we design several circuits that include arithmetic circuits, and simulate these circuits. Also, we compare our method with the LCC.

The rest of the paper is organised as follows: Section 2 introduces the LUT cascade emulator. Section 3 presents the logic simulator using the LUT cascade emulator. Section 4 describes the hybrid code generation. Section 5 shows experimental results. Finally, Section 6 concludes the paper.

II. LUT cascade emulator

A. LUT cascade

An **LUT cascade** is shown in Fig. 1, where multiple-output LUTs (**cells**) are connected in series to realize a

[1] When a primitive is simulated by an instruction of the processor, our simulator does not verify the bit precision problem. Thus, the verification for bit precision must be done by a different method.

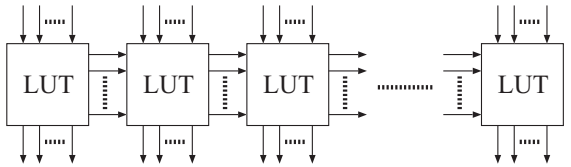

Fig. 1. LUT cascade.

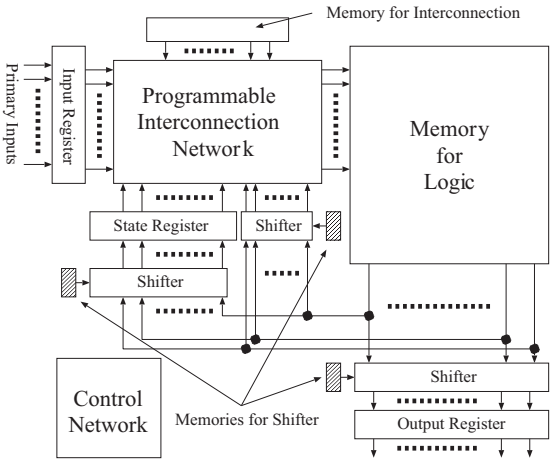

Fig. 2. LUT cascade emulator.

multiple-output function. The wires connecting adjacent cells are **rails**. Also, each cells may have external outputs in addition to the rail outputs. An LUT cascade is obtained by applying **functional decompositions** repeatedly to the BDD (**BDD_for_CF**) [2] that represents the multiple-output function [12].

B. LUT Cascade Emulator

The LUT cascade can realize only limited combinational circuits, once the parameters such as the number of cell inputs, the number of cells, and the number of rail outputs are fixed. Fig. 2 shows an **LUT cascade emulator** for a sequential circuit. Although it is slower than the LUT cascade, it is much more logically flexible than the LUT cascade.

The LUT cascade emulator stores the cell data of LUT cascades in the **Memory for Logic**; the address lines of cell are connected from inputs, state variables, and rail outputs of the preceding cell through the **Programmable Interconnection Network**; and the **Memory for Interconnection** stores data for the interconnections. The LUT cascade emulator reads the cell outputs from the memory for logic, and sends them to the **State Register** and the **Output Register** through **Shifters**; the **Memories for Shifter** store data for the shifters; the **Input Register** stores the values of the primary inputs; and the **Control Network** generates necessary control signals to obtain function values.

Example 2.1 *Fig. 4 shows an example of an LUT cascade. Fig. 5 shows an example of the memory for logic, for the cascade in Fig. 4. In Fig. 5,* **memory packing** *is done to reduce the necessary memory. Fig. 3 shows the pseudo-code to evaluate the LUT cascade emulator. (End of Example)*

```
1:  LUTCascadeEmulator(X = {x0, x1, x2, x3,
        x4, x5}, Mem){
2:      /* evaluate cell 0 */
3:      cellin ← {x0, x1};
4:      cellout ← Mem[cellin];
5:      rail ← 0x1 & cellout;
6:      pout ← ( 0x2 &cellout ) >> 1;
7:      /* evaluate cell 1 */
8:      cellin ← {rail, x2};
9:      cellout ← Mem[cellin];
10:     pout ← pout| (( 0x2 &cellout ) >> 1);
11:     /* evaluate cell 2 */
12:     cellin ← {x3, x4, x5};
13:     cellout ← Mem[cellin];
14:     pout ← pout| (( 0xD &cellout ) << 1);
15:     return pout;
16: }
```

Fig. 3. Pseudo-code for evaluating the LUT cascade emulator

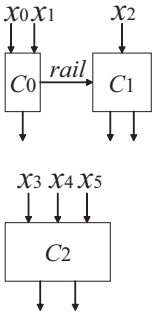

Fig. 4. An example of the LUT cascade.

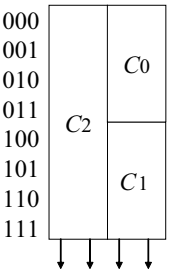

Fig. 5. An example of the memory for logic that stores cell data for Fig. 4.

The 3rd to 6th lines of Fig. 3 evaluate the cell 0. First, the 3rd line sets the input values for the cell 0. Next, the 4th line reads cell outputs from the memory for logic. After masking cell outputs, the 5th line reads the rail output. Also, the 6th line reads the primary outputs, and send them to the output register. To prepare the primary output for the next cell, it shifts the primary output by the predefined number of bits (10th and 14th lines). By repeating these operations, we can evaluate all cells in the LUT cascade. To evaluate the sequential circuit, we append the state variables and additional codes for updating the state variables, at the end of the pseudo-code.

III. Logic Simulation Using LUT Cascades

A. Scope of Our Simulator

Our simulator verifies RTL-level designs. The RTL-level simulation can be classified into two types: The **timing simulation** that considers the delay, and the **functional simulation** that ignores timing information. The logic simulators uses various methods: A **cycle-based method** can perform only the functional simulation, while an **event-driven method** can treat both types of simulations. Our hybrid simulator uses both the cycle-based and the event-driven methods. To simulate the special primitives (such

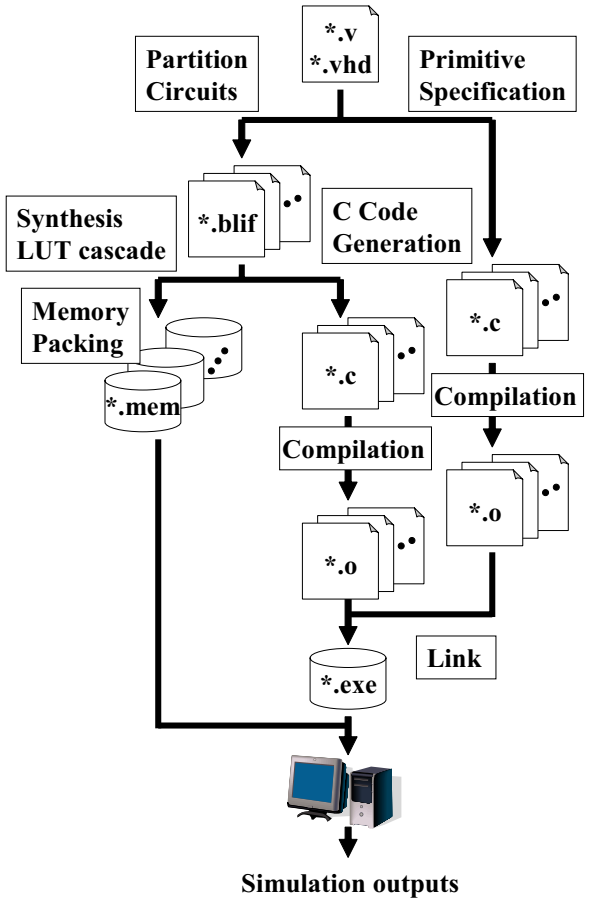

Fig. 6. Flow for simulator generation.

as memories and tri-state buffers) and the connections between partitioned circuits, our simulator uses the event-driven method. To simulate other parts, our simulator uses the cycle-based method. Thus, our simulator can perform only the functional simulation.

B. Flow for Simulator Generation

Fig. 6 illustrates the flow for simulator generation. After reading the design file (*.v,*.vhd) described in a gate-level HDL, the system converts it into partitioned BLIF (Berkeley Logic Interchange Format) files (*.blif) [14]. Also, it extracts special primitives which will be explained in Chapter IV. Next, it transforms each BLIF file into BDD_for_CFs to synthesize multiple LUT cascades. Then, the system stores cell data into the memory for logic, and generates memory data files (*.mem). At the same time, it generates the C code (*.c) that emulates LUT cascade emulators. Next, C codes for emulators and special libraries are compiled to generate object codes (*.o). Finally, object codes are linked to make the execution code (*.exe). The simulation is done by using the memory map files on the PC.

C. Partition of Circuits

An LUT cascade is obtained from the BDD by applying the functional decomposition repeatedly. In general, when the multiple-output function is represented by a single BDD, it cannot be stored into the memory on the PC, since the number of nodes is often too large. When the width for the BDD is large, it cannot be realized by the LUT cascade. Even if the multiple-output function can be realized by a single BDD, an excessive computation time is necessary to optimize the BDD. Thus, we partition the netlist into small modules, and realize each module by an LUT cascade emulator. The greedy algorithm described in [9] partitions the netlist to reduce the number of interconnections.

D. Generation of Memory Data

Our strategy is to reduce the sizes of cells for the LUT cascades. Also, to store the cell data for the LUT cascade, we use memory packing. As a result, the memory size is reduced. In this case, the number of cells increases, so the code size of the execution code increases. However, this reduces the simulation time, since the size of the execution code is much smaller than the size of memory data [9]. Smaller memory produces fewer cache misses.

E. Generation of Codes

C code is generated for each LUT cascade emulator that simulates the partitioned netlist. Each code is compiled into an object code. Then the object codes are linked to generate the execution code. Optimization of all the codes at the same time requires excessive time, about a half for the total simulation setup time[8]. Thus, each code is optimized independently to reduce the optimization time. Although the independent optimization increases the simulation time, this increase is much smaller than the time for the global optimization. As a result, the total computation time is reduced. Note that, the code for the LUT cascade emulator includes informations of the interconnections and the shift values, which correspond to the memory for interconnections and the memory for shifters.

IV. Hybrid Code Generation

A. Hybrid Logic Simulator

The number of rails of an LUT cascade is given by $\lceil \log_2 (\text{BDD width}) \rceil$. The LUT cascades are large for arithmetic circuits (e.g. multiplier and divider), since their BDDs and widths are large. The cycle-based simulator cannot simulate the special primitives such as memories and tri-state buffers. Thus, we build a hybrid logic simulator using both the cycle-based and the event-driven methods. The special primitives are simulated by the event-driven method, while the arithmetic circuits are simulated by instructions of the processor directly rather than the LUT cascades. Other parts are simulated using LUT cascade emulators. Special primitives and the arithmetic circuits are described by the special C code.

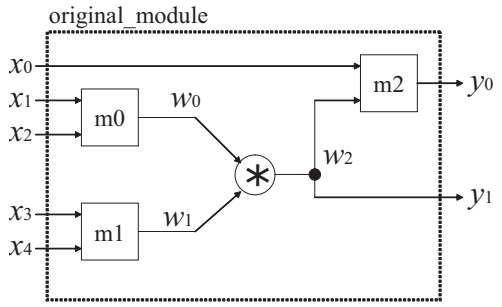

Fig. 7. An example of a module.

```
1:  original_module();
2:      input x0, x1, x2, x3, x4;
3:      output y0, y1;
4:      wire w0, w1, w2;
5:      m0(x1, x2, w0);
6:      m1(x3, x4, w1);
7:      assign w2 = w0 * w1;
8:      m2(x0, w2, y0);
9:      assign y1 = w2;
10: endmodule
```

Fig. 8. HDL code for Fig. 7.

B. Code Generation for Special Primitives

Fig. 6 illustrates the simulation flow with the special primitives. The special primitives to be converted into the functional library in the HDL code are described by comments. The system complies the codes for these primitives, and links with LUT cascade emulators. The following algorithm shows the method to extract the special primitive from the given HDL code.

Algorithm 4.1

1. Specify special primitives by the comments. Each special primitive is converted into a **functional library module**.
2. Attach the input and output signals to the special primitive module.
3. Modify the original module. Append outputs for the special primitive module. Also, append inputs for the special primitive module to produce the original module.
4. Interconnect the special primitive modules and the original modules. The system builds the modified module by the LUT cascade emulator.

Example 4.2 *Fig. 7 illustrates an example of a module. Fig. 8 shows the HDL code for Fig. 7. Let convert the multiplier primitive into the functional library using Algorithm 4.1. Specify the primitive by the comment (the 7th line in Fig. 8). To modify the original module, append the output signals w_{0_in} and w_{1_in}, that correspond to w_0 and w_1, respectively. Also, append the input signal w_{2_out} that corresponds to w_2. Fig. 9 illustrates the modified module with a multiplier functional library. Fig. 10 shows the HDL code for Fig. 9. (End of Example)*

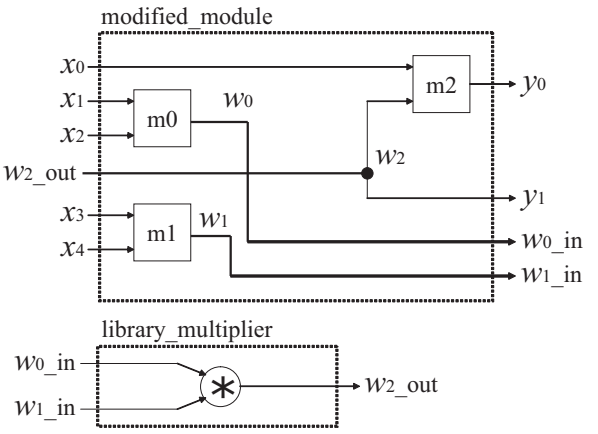

Fig. 9. Modules using the multiplier functional library module.

```
1:  modified_module();
2:      input x0, x1, x2, x3, x4;
3:      output y0, y1;
4:      input w2_out;
5:      output w0_in, w1_in;
6:      wire w0, w1, w2;
7:      m0(x1, x2, w0);
8:      m1(x3, x4, w1);
9:      assign w0_in = w0;
10:     assign w1_in = w1;
11:     m2(x0, w2_out, y0);
12:     assign y1 = w2_out;
13: endmodule;
```

Fig. 10. HDL code for Fig. 9.

V. Experimental Results

A. Environment for Experiment

We designed several circuits including arithmetic operations, and simulated these circuits on LUT cascade emulators and the LCC. In the experiments, we used an IBM PC/AT compatible machine, Pentium4 Xeon 2.8GHz, L1 Instruction Cache: 12μops, L1 Data Cache: 8KB, L2 Cache: 512KB, Memory: 4GByte, and OS: Redhad Linux 7.3. To produce the executable code, we used gcc compiler with optimization option -O3. We used a multiplier functional library [2] [3]. Designed circuits are as follows:

Vector norm function (NRM) computes $s = \sqrt{a^2 + b^2}$. The numbers of bits for inputs a, b are 8, and the number of bits for output s is 28 (sign 1bit, integer 16bit, and fraction 11bit). To compute the square root, we use a linear approximation method [7]. NRM uses three multipliers (two multipliers for the square operation and one multiplier for the liner approximation).

[2] In this experiment, the number of bits for the multiplier is at most 32, so the one instruction for the multiplication directly realizes the operation. However, when the number of bits for the multiplier exceeds 32, we must use several instructions to represent the multiplication.

[3] These simulators do not detect round-off error, truncation error, nor over-flow error, since an actual circuit often has a smaller number of bits.

Color Space Conversion (CSC) transforms the RGB signals (8bit×3) into the YCrCb signals (13bit×3) according to the equations:

$$\begin{aligned} Y &= 0.257R + 0.504G + 0.098B + 16 \quad (1) \\ Cb &= -0.148R - 0.291G + 0.439B + 128 \\ Cr &= 0.439R - 0.368G - 0.071B + 128 \end{aligned}$$

ITU-R BT.601 [4] recommendatory coefficients and additions of the quantization are used. CSC uses 9 multipliers.

Discrete Cosine Transformation (DCT) [15] performs the $8bit \times 8$ DCT operation according to the equations:

$$\begin{aligned} DCT_k &= \sum_{i=0}^{7} f_i c_{ki} \ (k = 0, 1, \ldots, 7) \quad (2) \\ c_{ki} &= \sqrt{\frac{2}{7}} \cos \frac{(2i+1)k\pi}{2 \times 7} \end{aligned}$$

The number of bits for outputs is $13bit \times 8$ (for jpeg and mpeg operations). The coefficients c_{ki} are precomputed, and stored in ROMs. Eight multipliers are used to compute each DCT_k. Thus, DCT uses 64 multipliers to compute from $k = 0$ to $k = 7$.

FIR filter (FIR) [13] is a low-pass filer. The number of taps is 17, and the cut-off frequency is 8000Hz. The number of input bits is 12 (44100Hz quantized), and the number of output bits is 19 (sign 1bit, integer 15bit, and fraction 3bit). This filter has the linear phase characteristics, so the number of multipliers can be reduced using the symmetric property. FIR_n for time n is represented by the equation:

$$FIR_n = \sum_{i=0}^{\frac{17-1}{2}} h_i(x_{n-1} + x_{n-17-i-1}) \quad (3)$$

The coefficient of h_i are precomputed and stored into ROMs. FIR uses 8 multipliers.

B. Comparison with LCC

Also in the LCC, we can use both partitioned circuits and the functional library. To analyze the performance, we compared four different simulation methods: The LUT cascade emulator without the functional library; the LCC without the functional library; the LUT cascade emulator with the multiplier functional library; and the LCC with the multiplier functional library. Note that, in all methods, we partitioned circuits. Table I compares four methods. In Table I, *Name* denotes the circuit name. Note that, when the multiplier functional library is used, *(lib)* is attached to *Name*. *Part* denotes the number of partitions; *Cell* denotes the number of cells; *E.in* denotes the total number of inputs (primary inputs, state inputs, and the wires connecting partitioned circuits) for cells; *Rail* denotes the total number of rails; *E.out* denotes the total number of outputs (primary outputs, state outputs, and the wires connecting partitioned circuits) for cells; *Code* denotes the execution code size; *Mem* denotes the size of memory for logic; *Literal* denotes the number of literals in the logical expression of LCC; *Setup* denotes the simulation setup time. *Setup* for the LUT cascade emulator includes time for circuit partition, for BDD generation, for memory packing, for code generation, and for compilation. *Setup* for the LCC includes time for circuit partition, for code generation, and for compilation. *Sim* denotes the simulation execution time for one million random test vectors.

Table I shows that the LUT cascade emulator requires about 1.2 times longer time than the LCC for simulation setup. However, the LUT cascade emulator is about 5.1 times faster than the LCC for simulation execution. *Code* for the LCC is larger than that of the LUT cascade emulator. In the LUT cascade emulator, circuits are transformed into both the code and the data, while in the LCC, all circuits are transformed into codes only. When the simulator uses the multiplier functional library, for the LUT cascade emulator, *Setup* is about 2.6 times shorter, and *Sim* is about 4.7 times faster. On the other hand, for the LCC, *Setup* is about 2.5 times shorter, and *Sim* is about 6.2 times faster.

The next expression estimates the simulation execution time of the LUT cascade emulator [9].

$$Est.Cas = E.in \times Cell + Cell + Rail + E.out. \quad (4)$$

The first term denotes the setup time of all the inputs of the cells; the second term denotes the access time to the memory for logic; the third term denotes the setup time for the rails; and the last term denotes the setup time for the registers. In Fig. 11, the right vertical axis denotes Sim.Cas (sec), the experimental value, and the left vertical axis denotes Est.Cas, the estimated number of operations. Also, we conjecture that *Literal* is almost proportional to the simulation time for the LCC. In Fig. 12, the right vertical axis denotes Sim.Cas (sec), the experimental value, and the left vertical axis denotes Est.LCC = *Literal*, the estimated number of literals. Figures. 11 and 12 show that the Sim.Cas and the Sim.LCC can be estimated from the Est.Cas and the Est.LCC, respectively. From Figs. 11 and 12, when the multiplier functional library is used, *Est.Cas* and *Est.LCC* are smaller than ones without libraries. By using the functional library, the multipliers are directly replaced with the instructions of a processor. It also reduces the $Literals$ and the size of the LUT cascade, so it also reduces *Est.Cas* and *Est.LCC*. As a result, the simulation is faster.

By using the functional library, we can simulate the special primitives, as well as reduce the simulation time.

C. Comparison with Commercial Tools

We compare our method with two commercial simulators: Super-FinSim [3] version 6.2.9 and ModelSim [6] Xilinx-Edition (XE) Starter 6.2g. The first one, ModelSim (XE) is an event-driven logic simulator, bundled with ISE. It supports functional simulation involving the 'zero-delay'. To obtain the evaluation time for the functional simulation, first, we generated a verilog code for a testbench

TABLE I
COMPARISON THE LUT CASCADE EMULATOR WITH THE LCC.

Name	Part	LUT cascade emulator								LCC			
		Cell	E.in	Rail	E.out	Code [KB]	Mem [KB]	Setup [sec]	Sim [sec]	Literal	Code [KB]	Setup [sec]	Sim [sec]
CSC	13	678	1554	533	405	49.6	13.9	3.5	3.6	4954	53.5	3.2	16.4
NRM	29	1085	2377	848	648	64.1	17.3	6.3	5.4	8349	84.2	6.2	31.3
FIR	29	1403	3158	1098	762	79.2	20.9	9.3	6.5	11190	104.6	8.4	44.3
DCT	38	1137	2678	796	966	75.2	17.4	8.4	6.0	10320	103.2	8.7	28.5
CSC (lib)	18	113	236	93	65	30.3	0.6	1.7	0.5	1299	12.9	1.7	3.7
NRM (lib)	10	289	592	243	140	32.9	3.2	1.7	0.7	1952	19.5	1.7	4.4
FIR (lib)	25	392	901	314	220	51.1	1.0	4.1	2.2	3562	35.6	4.1	6.6
DCT (lib)	17	329	735	258	248	38.2	7.0	2.8	1.6	2926	29.2	2.8	5.0
ratio								1.0	1.0			0.8	5.1

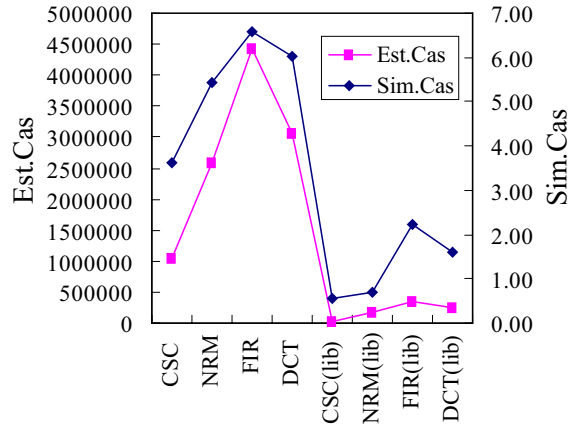

Fig. 11. Simulation execution time for the LUT cascade emulator.

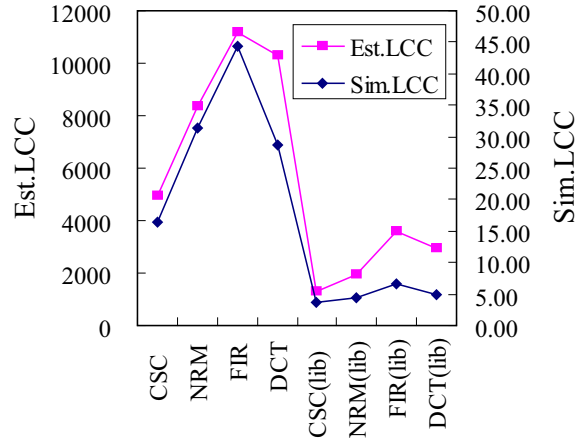

Fig. 12. Simulation execution time for the LCC.

including 10^6 test vectors. Then, we compiled the verilog codes, representing benchmark circuit and testbench, using a command 'vlog [benchmark name.v]' with option '+notimingcheck' and 'no_notifier'. Next, we evaluated the simulated time using commands which are 'vsim -c' and 'run 1000000'. The second one, Super-FinSim supports both the enhanced cycle simulation (ECS) and the event-driven simulation at the same time. To compare our method with the ECS, we generated the testbench similar to ModelSim, and we complied verilog codes using a command 'finvc' with option '+delay_mode_zero', '-dsm com', '-ol 1', '-acc', '-fastgate', '+notimingcheck', and '+no_notifier'. Then, we built an execution file using the command 'finbuild', and evaluated the simulation time. We used Microsoft Visual C++ version 6.0 for Super-FinSim. In the experiments, we used an IBM PC/AT compatible machine, AMD Athlon 64 FX-60, L1 Instruction Cache: 64KB, L1 Data Cache: 64KB, L2 Cache: 1024KB, Memory: 3GByte, and OS: Windows XP Professional SP2. To realize the LUT cascade emulator for Linux on Windows, we used gcc compiler version 3.2 on the cygwin.dll version 1.3.22 which emulates Linux API using Windows API.

TABLE II
COMPARISON OF SIMULATION TIME.

Name	Simulation time			Ratio	
	FinSim	ModelSim	LUT Cascade Emulator	v.s.FinSim	v.s.ModelSim
CSC	3.5	23.3	0.5	7.0	46.6
NRM	3.8	25.2	0.7	5.4	36.0
FIR	5.1	24.4	2.2	2.3	11.0
DCT	4.1	25.0	1.6	2.5	15.6

Table II compares the results. *Simulation time* denotes the actual simulation time including the time for reading and writing 10^6 vectors; and *Ratio* denotes that of the simulation execution time (Super-FinSim / LUT cascade emulator) and (ModelSim / LUT cascade emulator), respectively. Although ratios for our new method are smaller than the previous method [8], it can realize a wide range of logic functions that cannot be realized by the previous one.

Table II shows that the LUT cascade emulator is 11.0-46.6 times faster than ModelSim, and 2.3-7.0 times faster than FinSim. Note that, the performance for ModelSim XE Starter is restricted to 20% of Professional edition. Our method is faster than ModelSim, even if this restriction is considered.

VI. Conclusion

This paper presented a hybrid logic simulator using both cycle-based and event-driven methods. Special primitives are simulated by the event-driven method, while other parts are simulated by the cycle-based method. A large circuit is partitioned into smaller ones. Then each part is realized by an LUT cascade emulator. Each emulator are connected by wires generated by the system. As for simulating multipliers, we used the instruction of the processor rather than an LUT cascade. With this technique, we could reduce code size as well as simulation time. Our experiments showed that proposed simulator is effective for circuits including arithmetic operations.

VII. Acknowledgements

This research is partly supported by Japan Society for the Promotion of Science (JSPS) and Grant in Aid for Scientific Research.

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design, Wiley-IEEE Press*; Rev.Print edition, Sept. 1994.

[2] P. Ashar, and S. Malik, "Fast functional simulation using branching programs," *Int'l Conf. on Computer-Aided Design (ICCAD95)*, Nov.1995, pp.408-412.

[3] Fintronic USA, Inc.: http://www.fintronic.com/

[4] International Telecommunication Union: http://www.itu.int/

[5] P. C. McGeer, K. L. McMillan, A. Saldanha, A. L. Sangiovanni-Vincentelli, and P. Scaglia, "Fast discrete function evaluation using decision diagrams," *Int'l Conf. on Computer-Aided Design (ICCAD95)*, pp.402-407, Nov. 1995.

[6] Mentor Graphics Corporation: http://www.model.com/

[7] S. Nagayama, T. Sasao, and J. T. Butler, "Programmable numerical function generators based on quadratic approximation: Architecture and synthesis method," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC2006)*, Yokohama Jan. 2006, pp. 378-383.

[8] H. Nakahara, T. Sasao and M. Matsuura, "A fast logic simulator using an LUT cascade emulator," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC2006)*, Yokohama Jan. 2006, pp.466-465.

[9] H. Nakahara, T. Sasao and M. Matsuura, "A PC-based logic simulator using a look-up table cascade emulator," *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E89-A, No.12, Dec. 2006, pp.3471-3481, Special Section on VLSI Design and CAD Algorithms.

[10] T. Sasao, H. Nakahara, M. Matsuura and Y. Iguchi, "Realization of sequential circuits by look-up table ring," *The 2004 IEEE Int'l Midwest Symp. on Circuits and Systems (MWSCAS2004)*, Hiroshima, July 25-28, 2004, pp.I:517-I:520.

[11] T. Sasao, M. Kusano, and M. Matsuura, "Optimization methods in look-up table rings," *13th Int'l Workshop on Logic and Synthesis (IWLS2004)*, June 2-4, 2004, Temecula, California, USA, pp.431-437.

[12] T. Sasao, Y. Iguchi, M. Matsuura, "LUT cascades and emulators for realizations of logic functions," *7th Int'l Symp. on Representations and Methodology of Future Computing Technologies(RM2005)*, Tokyo, Japan, Sept. 5 - Sept. 6, 2005, pp.63-70.

[13] T. Sasao, Y. Iguchi, T. Suzuki, "On LUT cascade realizations of FIR filters," *8th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD2005)*, Porto, Portugal, Aug. 30 - Sept. 3, 2005, pp.467-474.

[14] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," *Technical Report UCB/ERL M92/41, U.C. Berkeley*, May 1992.

[15] Xilinx, "Video compression using DCT," *Xilinx Application Note:* http://www.xilinx.com/