A Fast Logic Simulator Using a Look Up Table Cascade Emulator

Hiroki Nakahara Tsutomu Sasao Munehiro Matsuura

Depertment of Computer Science and Electronics
Kyushu Institute of Technology, Iizuka 820-8502, Japan

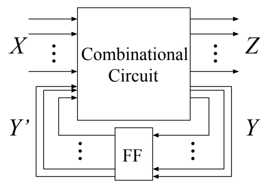
Abstract— This paper shows a new type of a cycle-based logic simulation method using a Look-Up Table (LUT) cascade emulator. The method first transforms a given circuit into LUT cascades through BDD (Binary Decision Diagram). Then, it stores LUT data to the memory of an LUT cascade emulator. Next, it generates the C code representing the control circuit of the LUT cascade emulator. And, finally, it converts the C code into the execution code. This method is compared with a Levelized Compiled Code (LCC) simulator with respect to the simulation time and setup time. Although we used standard PC to simulate the circuit, experimental results show that this method is 12-64 times faster than the LCC.

I. INTRODUCTION

With the increase of the integration of LSIs, the time for the verification of the design increases. Thus, high-speed logic simulators are needed.

Logic simulators can be roughly divided into two types: event-driven simulators and cycle-based simulators. In an event-driven simulator, only the logic gates whose input signal change are evaluated. On the other hand, in a cycle-based logic simulator, the operation order of gates are determined statically beforehand, and all the logic values of the gates are evaluated for each clock cycle. Although the cycle-based logic simulator does not perform the timing verification, it is often faster than the event-driven simulator.

An **LCC** [1] is a kind of a cycle-based logic simulator using a general-purpose CPU. An LCC generates a program code for each gate of a logic circuit, and evaluates the circuit in a topological order from the inputs towards the outputs. An event-driven simulator that emulates only logic gates whose outputs change has been developed [16]. It is at most two times faster than the LCC. In this paper, we will present a cycle-based logic simulator using an LUT cascade emulator. An LUT cascade emulator [2] consists of a control part, memories, and registers. Each register is connected to a programmable interconnection circuit, and the LUT cascade emulator evaluates the logic circuit stored in the memory. Murgai-Hirose-Fujita [11] also developed a logic simulator using large memories. Their method first converts a given circuit into a random logic network of single-output LUTs, then stores them in the memory, and finally evaluates the circuit by an event-driven logic simulator implemented by a hardware accelerator. In our method, we first convert the given circuit into a cascade rather than random logic, so the control part is simpler than Murgai-Hirose-Fujita's method. Also, our method uses multiple-output LUTs rather than single-output LUTs. In this paper, we consider the software-based logic simulation system where the LUT cascade emulator is simulated on a PC. Compared with the hardware-based logic simulator, logic simulator using a standard PC is much cheaper, and can be enhanced with the improvement of the performance of PCs.



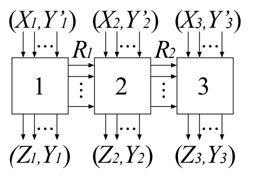


Fig. 1. A model for a sequential circuit. Fig. 2. LUT cascade.

II. LUT CASCADE EMULATOR

A. LUT Cascade

Fig. 1 shows a model for a sequential circuit, where X denote inputs, Z denote outputs, Y denotes the inputs to flip-flops, Y' denotes the outputs of flip-flops, and $|Y|$ denotes the number of state variables. We first consider an **LUT cascade** [3] that realizes the combinational part of the sequential circuit, then consider the LUT cascade emulator that emulates the LUT cascade.

An LUT cascade is shown in Fig. 2, where multiple-output LUTs (**cells**) are connected in series to realize a multiple-output function. The wires connecting adjacent cells are called **rails**. Also, each cells may have external outputs in addition to the rail outputs. In this paper, X_i denotes the **external inputs** to the i-th cell; Y_i' denotes the **state inputs** to the i-th cell; Z_i denotes the **external outputs** of the i-th cell; Y_i denotes the **state outputs** of the i-th cell; R_{i-1} denotes the **rail inputs** to the i-th cell; and R_i denotes the **rail outputs** from the i-th cell. We can obtain the LUT cascades by applying **functional decomposition** repeatedly to the BDD that represents the multiple-output function [4].

Definition 2.1 *Let* $\vec{X} = (x_1, x_2, \ldots, x_n)$ *be the input variables,* $\vec{Y} = (y_1, y_2, \ldots, y_m)$ *be the output variables, and* $\vec{f} = (f_1(\vec{X}), f_2(\vec{X}), \ldots, f_m(\vec{X}))$ *be the corresponding out-*

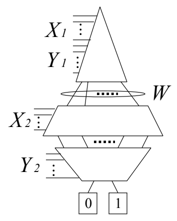


Fig. 3. BDD_for_CF.

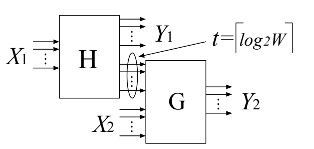


Fig. 4. Functional decomposition.

put functions. **The characteristic function of the multiple-output function** *is* $\vec{\chi}(\vec{X},\vec{Y}) = \bigwedge_{i=1}^{m}(y_i \equiv f_i(\vec{X}))$.

The characteristic function of an n-input m-output function is a two-valued logic function with $(n+m)$ inputs. It has input variables x_i $(i = 1,2,\ldots,n)$, and output variables y_j for output f_j. Let $B = \{0,1\}$, $\vec{a} \in B^n$, $\vec{F} = (f_1(\vec{a}), f_2(\vec{a}), \ldots, f_m(\vec{a})) \in B^m$, and $\vec{b} \in B^m$. Then, the characteristic function satisfies the relation

$$\vec{\chi}(\vec{a},\vec{b}) = \begin{cases} 1 & (\text{when } \vec{b} = \vec{F}(\vec{a})) \\ 0 & (\text{otherwise}) \end{cases}$$

Definition 2.2 A support variable *of a function f is a variable on which f actually depends.*

Definition 2.3 *[5]* **The BDD_for_CF** *of a multiple-output function $\vec{f} = (f_1, f_2, \ldots, f_m)$ is the ROBDD [10] for the characteristic function $\vec{\chi}$. In this case, we assume that the root node is in the top of the BDD, and the variable y_i is below the support variable of f_i, where y_i is the variable representing f_i.*

Definition 2.4 The width of the BDD_for_CF *at height k is the number of edges crossing the section of the graph between x_k and x_{k+1}, where the edges incident to the same nodes are counted as one. Also, in counting the width of the BDD_for_CF, we ignore the edges that incident to the constant 0 node.*

Let X_1 and X_2 be sets of input variables, Y_1 and Y_2 be sets of output variables, (X_1, Y_1, X_2, Y_2) be the variable ordering of a BDD_for_CF for the multiple-output function $\vec{f} = (f_1, f_2, \ldots, f_m)$, and W be the width of the BDD_for_CF at the height (X_1, Y_1) in Fig. 3. By applying functional decomposition to $\vec{f}$, we obtain the network in Fig. 4, where the number of lines connecting two blocks is $t = \lceil \log_2 W \rceil$ [4].

Theorem 2.1 *[5] Let μ_{max} be the maximum width of the BDD_for_CF that represents an n-input logic function $\vec{f}$. If $u = \lceil \log_2 \mu_{max} \rceil \leq k-1$, then $\vec{f}$ can be realized by a circuit shown in Fig. 4, where $|X_1| = k$. By applying functional decompositions $s-1$ times, we have the cascade having the structure of Fig. 2.*

B. LUT Cascade Emulator

Fig. 5 shows an **LUT cascade emulator** for a sequential circuit.

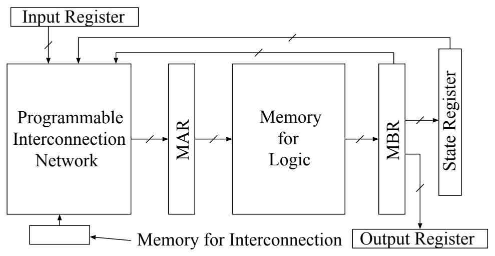


Fig. 5. LUT cascade emulator.

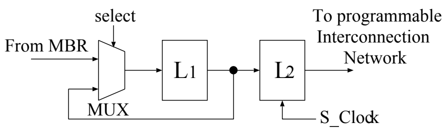


Fig. 6. Double rank flip-flop.

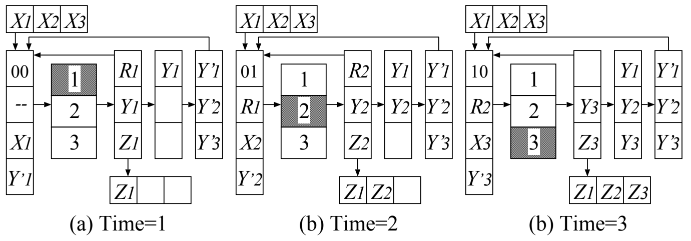


Fig. 7. Emulation of a sequential circuit

An LUT cascade emulator stores the cell data of an LUT cascade in the **memory for logic**. The address of cell data is calculated from inputs, state variables, and rail outputs of the preceding cell. The LUT cascade emulator reads the cell outputs from the memory for logic, and send them to the **State Register** and the **Output Register**. The **Input Register** stores the values of the primary inputs; the **MAR** (Memory Address Register) stores the address of the memory; the **MBR** (Memory Buffer Register) stores the outputs of the memory; the **Programmable Interconnection Network** connects the input register, the state register, and the MAR, also it connects the MBR and the MAR; the **Memory for Interconnection** stores data for the interconnections; and the **Control Network** generates necessary signals to obtain functional values.

To emulate a sequential circuit, the LUT cascade emulator stores state variables and output variables in the registers. Fig. 6 shows the **Double-Rank Flip-Flop** for the state register and the output register, where L_1 and L_2 are D-latches. Set the select signals to high when all the cells in a cascade are evaluated, and send the values into L_1 latches. When all the cascades are evaluated, the values of the state variables are sent to L_2 latches. This can be done by adding a pulse to S_Clock.

Example 2.1 *Fig. 7 illustrates the emulation of the sequential circuit whose combinational part realizes the LUT cascade in Fig. 2.*

At $Time = 1$, to evaluate Cell 1, the two most significant bits of the address are set to (0,0) to specify Page 1. Also, the inputs to Cell 1 X_1 and Y_1' are set to the lower address bits through the programmable interconnection network. By reading Page 1, the outputs of Cell 1 are sent to MBR. For the outputs that become the primary output Z_1, store it in the output register, while for the output that becomes the state output Y_1, store it in the state register (Fig. 7(a)).

At $Time = 2$, to evaluate Cell 2, the two most significant bits of the address are set to (0,1) to specify Page 2. Also, the outputs of Cell 1, R_1 are connected to the middle address bits, and the input variables of Cell 2 X_2 and Y_2' are set to the lower address bits through the programmable interconnection network. By reading Page 2, the outputs of Cell 2 are sent to MBR. For the output that becomes the primary output Z_2, store it in the output register, while for the output that becomes the state output Y_2, store it in the state register (Fig. 7(b)).

At $Time = 3$, to evaluate Cell 3, the two most significant bits of the address are set to (1,0) to specify Page 3. Also, the outputs of Cell 2, R_2 are connected to the middle address bits, and the input variables of Cell 3 X_3 and Y_3' are set to the lower address bits through the programmable interconnection network. By reading Page 3, the outputs of Cell 3 are sent to MBR. For the output that becomes the primary output Z_3, store it in the output register, while for the output that becomes state output Y_3, store it in the state register (Fig. 7(c)).

When all the cells of cascades are evaluated, Control Network sends a pulse to S_Clock of the state register and the output register, and the values of state outputs Y_1, Y_2, Y_3 are sent to the L_2 latches. Also, the values of the output register Z_1, Z_2, Z_3 are sent to the primary outputs. (End of Example)

III. Synthesis of the LUT Cascade Emulator

A. Partition of the outputs

When the number of outputs is large, we partition the outputs into groups, and realize a cascade for each group independently. Usually, the BDD_for_CF for all the outputs are too large to construct. Even if the BDD_for_CF is constructed, it can be too large to be realized by an LUT cascade. Also, constructing a single BDD_for_CF for all the outputs is inefficient, since the optimization of a large BDD_for_CF is time consuming.

In order to construct as small BDD_for_CFs as possible, we partition the outputs so that each group has a small number of support variables.

Definition 3.5 *Let $F = \{f_1, f_2, \ldots, f_m\}$ be the set of the outputs functions, $G \subseteq F$, and $f_i \in F - G$. Then, the* **similarity** *of the output f_i with G is defined as follows:*

$$Similarity(i, G, F) \quad = \quad |Sup(f_i) \cap Sup(G)|, \tag{1}$$

where $Sup(F)$ denotes the set of support variables of F.

Algorithm 3.1 *(Partition the Outputs and Construction of BDD_for_CF)*
Let $F = \{f_1, f_2, \ldots, f_m\}$ be the set of m logic functions, $\mathcal{Z} = \{G_1, G_2, \ldots, G_r\}$ be the set of subset of output functions after partitioning, r be the number of partitions, and Th_node be the threshold of the number of nodes.

```
1. $\mathcal{Z} \leftarrow \phi, r \leftarrow 0$.
2. While $F \neq \phi$, do Steps (a)-(d).
    (a) $Node \leftarrow 0, G_r \leftarrow \phi$.
    (b) While $Node \leq Th\_node$, $F \neq \phi$, and $G_r$ is cascade realizable, do Steps i-iii.
        i. Select $f_i$ with the maximum $Similarity(i, G_r, F)$. If $G_r = \phi$, then select $f_i$ that has the largest support.
        ii. $G_r \leftarrow G_r \cup \{f_i\}, F \leftarrow F - \{f_i\}$.
        iii. Construct BDD_for_CF that realizes $G_r$, and $Node \leftarrow (the\ number\ of\ nodes)$.
    (c) If $G_r$ is not cascade realizable, then $G_r \leftarrow G_r - \{f_i\}, F \leftarrow F \cup \{f_i\}$.
    (d) $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{G_r\}, r \leftarrow r + 1$.
3. Terminate.
```

This method merges outputs into a group while a cascade is realizable and the number of nodes in the BDD is equal to or less than the threshold. Algorithm 3.1 partitions the outputs into the groups so that the resultant BDDs are small enough.

B. Memory Packing

By Alogrithm 3.1, a given multiple-output function is represented by a set of BDD_for_CFs. Then, the LUT cascades are constructed, and LUT data is allocated into the memory of the LUT cascade emulator.

Example 3.2 *Fig. 8 shows the LUT cascade consisting of 4-input cells. Fig. 9(a) illustrate the memory map of cell data, where the memory has 6-bit address inputs, and each word consists of four bits. The dark parts in the figure are unused, and P_i denotes the page number. (End of Example)*

In Example 3.2, each cell data are stored in a separate page of the memory. The data of a cell must be stored in the same page, and must be read simultaneously. However, if there are any extra space in the same page, multiple cell data can be stored in the same page. This method to reduce the memory area is called **memory-packing** [6].

Example 3.3 *In Fig. 9(a), by storing the cell data r_5 and z_1 to Page 1, we have the memory map in Fig. 9(b), where a half of the memory is enough to store all the data. (End of Example)*

IV. Logic Simulation using an LUT Cascade Emulator

A. Generation of the execution code for the simulation

Fig. 10 shows the data flow of the logic simulation system using LUT cascade emulator.

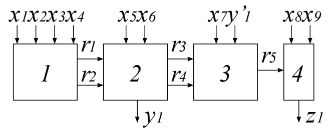


Fig. 8. Example of LUT cascade.

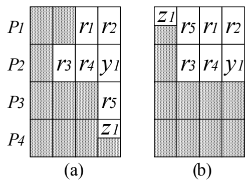


Fig. 9. Example of memory-packing.

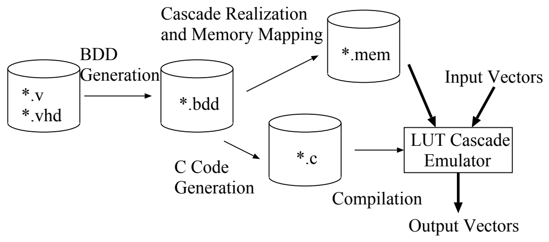


Fig. 10. Flow of data for the logic simulation system using LUT cascade emulator.

First, it converts the Verilog-VHDL code describing the given circuit into shared-BDDs [9]. Then, it reduces the number of nodes of BDDs by changing variable orders [7]. Next, it generates the LUT cascades from BDDs using the functional decomposition described in Chapter 2, and it maps them to the memory of the LUT cascade emulator. Also, it generates the C code that simulates the control circuit of the LUT cascade emulator. Next, it complies the C code into the execution code for simulation of the LUT cascade emulator. And, finally the simulator evaluates the output of the given circuit, by using the memory map of the LUT cascade emulator.

B. Program code that simulates the LUT cascade emulator

This system generates the program code that describes the following operations.

Step 1 Set the input register, and initialize the state register.
An input value is set to the input register. Also, the value of the state register is initialized.

Step 2 Evaluate of each cell.

Step 2.1 Simulate the programmable interconnection network.
The address of the memory for logic is generated from the values of the input register, the state input register, the MBR, and a page address.

Step 2.2 Read the memory for logic.
The content of the memory for logic is read using the address generated by Step 2.1.

Step 2.3 Distribute output values of the memory for logic.
The values read by Step 2.2 are sent to the output register and the state register.

Step 3 Perform the state transitions.
The output values of the state register is updated by S_Clock.

Assigning each memory output to each register consumes CPU time. Fortunately, the memory outputs are stored in the order of primary outputs, state outputs, and rail outputs. For a 32-bit processor, we can evaluate up to 32 outputs at a time. To obtain required outputs, we shift the memory outputs covered by a mask, and assign to a 32-bit variable. In this way, we can evaluate the multiple outputs simultaneously. Also, there is an additional merit for performing the state transition. Let $|Y|$ be the number of state variables for given logic function, then the number of evaluations for the state transition is $\left\lceil \frac{|Y|}{32} \right\rceil$ for a 32-bit machine.

Since cascades have much fewer signal lines than the original circuit, the compilation time for cascades are much shorter than that of the conventional LCC method.

C. Analysis of Simulation Time

When the LUT cascade emulator is implemented on a dedicated hardware [2], the evaluation time is proportional to the number of cells. However, when the LUT cascade emulator is implemented on a standard PC, we need extra time since the inputs and outputs of a cell are evaluated sequentially.

To do high-speed simulation for the LUT cascade emulator on a PC, we have to consider two different objects:

a. Reduction of the number of cells.
This can be done by increasing the number of inputs of each cell. However, the increase of the number of inputs of each cell also increases the evaluation time per cell.

b. Reduction of the number of cell inputs.
This decreases the evaluation time per cells, but increases the number of cells.

To find the best strategy, we did following experiments. We implemented 10 MCNC benchmark functions [8] on the LUT cascade emulator. By changing the maximum number of inputs for cells, we obtained the average number of cell inputs, the number of cells, and the execution time of the LUT cascade emulator. Fig. 11 shows the experimental results, where the horizontal axis denotes the maximum number of cell inputs; 0 denotes the lower bound on the maximum number of inputs of cells, that is $\lceil \log_2 \mu_{max} \rceil + 1$; the vertical axis denotes the ratios of the number of cells, the number of the average cell inputs, and simulation time. We set 1.00 to the ratios when the number of cell inputs is $\lceil \log_2 \mu_{max} \rceil + 1$.

Fig. 11 shows that the simulation time increases with the number of cell inputs. To compute the address of the memory for logic, we need CPU time. This CPU time increase with the number of inputs of a cell. Therefore, our strategy is to reduce the number of cell inputs in the LUT cascade emulator.

V. Experimental Results

A. Comparison with LCC

We simulated selected MCNC benchmark functions by LUT cascade emulator and LCC on a same PC. Table I shows the

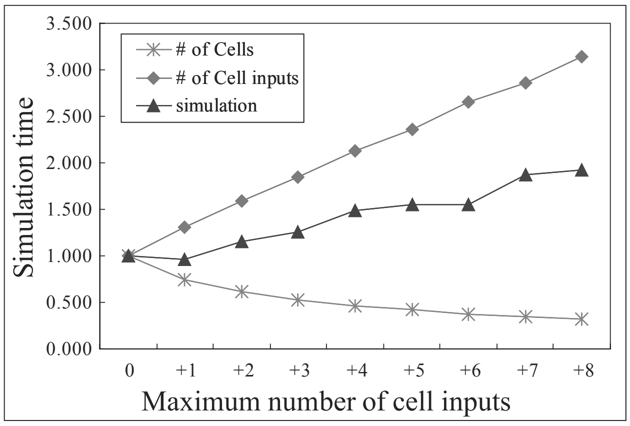


Fig. 11. Relation between the maximum number of cell inputs and simulation time.

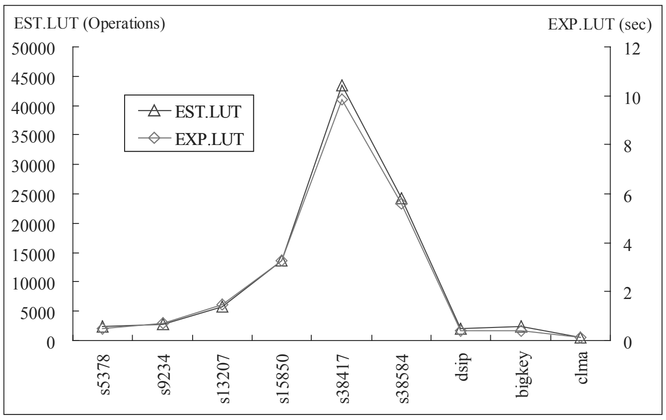


Fig. 12. Simulation time for LUT cascade emulator.

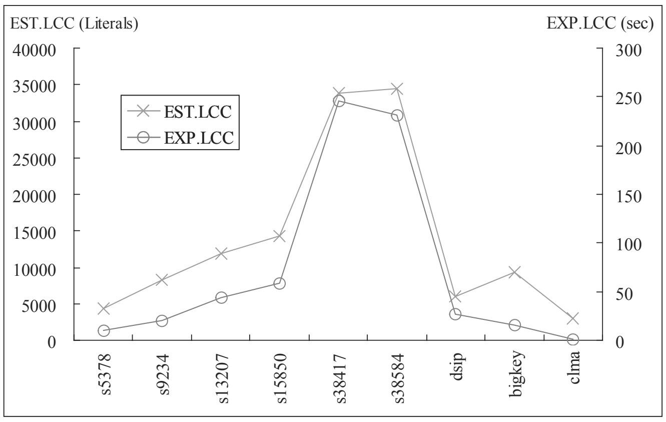


Fig. 13. Simulation time for LCC.

experimental results. *Name* denotes the name of benchmark function; *In* denotes the number of inputs; *Out* denotes the number of outputs; *State* denotes the number of state variables; *Cas* denotes the number of LUT cascades; *Cell* denotes the total number of cells; and *Mem* denotes the amount of memory (Mega Bits). Also, *EXT.in* denotes the average number of external inputs to cells; *P.out* denotes the total number of cells with external output(s); and *S.out* denotes the total number of cells with state output(s). *Sim* denotes the evaluation time (sec). In order to obtain the raw evaluation time for the simulation, we generated the one million random test vectors, and evaluated the time excluding the time for reading and writing vectors. *Setup* denotes the setup time (sec) for a simulation. *Setup* of LCC is the time for the C code generation and the compilation, while *Setup* of the LUT cascade emulator is the time for BDD generation, LUT cascades synthesis, memory mapping, C code generation, and the compilation. *Literals* denotes the total number of literals in expressions of lines of the C code generated by the LCC. *Ratio* denotes that of the simulation setup time or that of the simulation execution time (LCC/LUT cascade emulator). To produce the executable code for LCC, we used gcc compiler with optimization option -O3. Also, we generated program codes for LUT cascade emulator, and compiled them with the same conditions as LCC. In the experiments, we used an IBM PC/AT compatible machine, Pentium4 Xeon 2.8GHz, L1 Cache: 8KB, L2 Cache: 512KB, Memory: 4GByte, and OS: Redhad (Linux 7.3). For the benchmark function *clma*, it has many redundant primary inputs, primary outputs, and logic gates. For pre-processing, we simplified the Verilog-VHDL descriptions of benchmark functions by using Quartus II version 5.0 [13]. Note that, the time for the pre-processing is not included in the setup time.

Table I shows that the LUT cascade emulator is 12 - 64 times faster than the LCC. Also, the setup speed for the LUT cascade emulator is 1.20 - 5.22 times faster than the LCC, except for s5378 and *clma*.

The number of operations in the LUT cascade emulator is estimated as follows:

$$\begin{aligned} EST.LUT &= EXT.in \times Cell \\ &\quad + Cell + P.out + S.out + Rail, \qquad (2) \end{aligned}$$

where $Rail = Cell - Cas$. The first term of expression (2) corresponds to the setup time of all the external inputs of the cells; the second term corresponds to the access time to the memory for logic; the third term corresponds to the setup time for the output register; the fourth term corresponds to the setup time for the state register; and the last term corresponds to the setup time for the rail inputs.

In Fig. 12, the right vertical axis denotes the experimental value EXP.LUT (sec), and the left vertical axis denotes the estimated number of operations EST.LUT. Also, we conjecture that *Literals* is proportional to the simulation time for LCC. In Fig. 13, the right vertical axis denotes the experimental value EXP.LCC (sec), and the left vertical axis denotes the estimated number of literals EST.LCC = *Literals*. Figs. 12 and 13 show that the EXP.LUT and EXP.LCC can be estimated from EST.LUT and EST.LCC, respectively.

In Figs. 12 and 13, note that EXP.LUT is smaller than EXP.LCC, while EST.LUT (number of operations) is larger than EST.LCC (number of literals) for some functions. This may look strange. To see the reason, we converted the C codes

TABLE I
RESULTS OF REALIZATION OF BENCHMARK FUNCTIONS.

Name	In	Out	State	LUT cascade emulator								LCC			Ratio	
				Cas	Cell	Mem [Mbit]	EXT.in	P.out	S.out	Setup [sec]	Sim [sec]	Literals	Setup [sec]	Sim [sec]	Setup	Sim
s5378	35	49	164	40	543	0.82	2.39	105	28	14.59	0.49	4424	10.26	10.46	0.70	21.35
s9234	36	39	211	44	599	0.91	2.63	26	120	14.68	0.71	8220	41.01	20.64	2.79	29.07
s13207	62	152	638	93	1245	3.28	2.27	109	390	31.09	1.46	11954	83.04	43.76	2.67	29.97
s15850	77	150	534	105	3370	8.95	1.95	115	338	79.25	3.25	14328	115.72	58.71	1.46	18.06
s38417	28	106	1636	389	9411	45.36	2.55	60	964	763.07	9.87	33769	917.40	245.63	1.20	24.89
s38584	38	304	1426	270	5118	16.90	2.55	232	956	159.09	7.61	34485	830.27	230.76	5.22	30.33
dsip	229	197	224	45	473	6.60	1.96	108	115	10.90	0.42	5959	26.39	27.08	2.42	64.48
bigkey	263	197	224	48	541	3.76	1.97	171	121	11.74	0.42	9262	27.58	16.26	2.35	38.71
clma	101	81	33	9	45	0.16	3.28	27	21	3.59	0.11	2994	2.87	1.37	0.79	12.45

of the LUT cascade emulator and the LCC into the assembly-codes, and analyzed them. The size of the assembly-code for the LCC is several times larger than EST.LCC. This is because the LCC compiler generates extra codes to evaluate the negative literals and the logic gates, and to produce the output signals. On the other hand, the size of the assembly-code for LUT cascade emulator does not depend on EST.LUT, and is much smaller than EST.LUT. In the LCC, it's operands frequently move between register and memory. For the gate with fanouts, the LCC stores the output values of the gate into a variable temporarily, and uses it as the input of two or more gates. On the other hand, the LUT cascade emulator uses only the rail values in the single register variable. Therefore, only the input and output register and the memory for logic requires memory references. Experimental results show that the simulator based on an LUT cascade emulator is 12-64 times faster than the LCC. One reason for this is the difference of the representations: the cascade has many fewer signals than the random logic network. Another reason is due to the CPU architecture of the PC. The access time of the data in the main memory is about 200 times longer than one in the L1 cache. So, the CPU time heavily depends on the frequency of cache miss. In the case of the LCC simulator, the circuit data and control are mixed, and the instruction data is too large to be stored in the data cache. On the other hand, in the case of an LUT cascade emulator, the cascade data and control are separated. Control data is in the instruction cache, while the cascade data is in the data cache. Thus, we can expect fewer cache miss in the LUT cascade emulator.

B. Comparison with Commercial Tools

We compare the our method with two commercial simulators: Super-FinSim [14] version 6.2.9 and ModelSim [15] Altera-Edition (AE) 6.0c. ModelSim (AE) is an event-driven logic simulator, bundled with Quartus II. It supports functional simulation involving the 'zero-delay'. To obtain the evaluation time for the functional simulation, first, we generate a verilog code for a testbench including 10^4 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 evaluate the simulated time using commands which are 'vsim -c' and 'run 10000'. Super-FinSim supports both the enhanced cycle simulation (ECS) and the event-driven simulation at the same time. To compare the 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, Pentium4 3.06GHz, L1 Cache: 8KB, L2 Cache 512KB, Memory 2GByte, 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
RESULTS OF COMPARISON WITH COMMERCIAL TOOLS.

Name	Simulation time			Ratio	
	FinSim	ModelSim	LUT Cascade Emulator	v.s.FinSim	v.s.ModelSim
s5378	1.45	35.77	0.09	16.11	397.44
s9234	1.73	44.72	0.11	15.73	406.55
s13207	6.59	46.50	0.17	38.76	273.53
s15850	14.84	68.63	0.58	25.59	118.33
s38417	17.93	72.70	3.28	5.47	22.16
s38584	58.21	155.84	1.76	33.07	88.55
dsip	14.31	30.42	0.16	89.44	190.13
bigkey	2.99	37.28	0.08	37.38	466.00
clma	41.13	88.86	0.09	457.00	987.33

Table II compares the results. *Simulation time* denotes the actual simulation time including the time for reading and writing 10^4 vectors; and *Ratio* denotes that of the simulation execution time (Super-FinSim / LUT cascade emulator) and (ModelSim / LUT cascade emulator), respectively.

Table II shows that the LUT cascade emulator is 22.16-987.33 times faster than ModelSim, and 5.47-457.00 times faster than FinSim.

Especially, for benchmark *clma*, our method is hundreds times faster than commercial tools. *clma* is a special benchmark circuit with redundant input-and-output signals and redundant gates. In order to perform a high-speed simulation, our method converted the circuit into BDDs to remove these

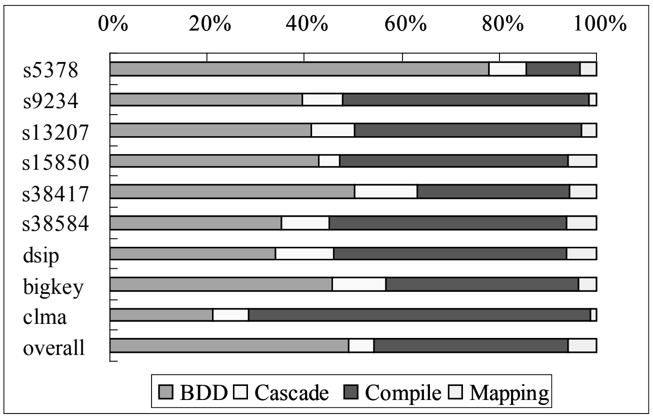


Fig. 14. Percentage of each process for setup time.

redundancy. However, ModelSim and FinSim used original circuits, so the redundancy of circuits affected the simulation time.

C. Setup Time for the LUT Cascade Emulator

Fig. 14 shows the percentage of each process of the setup time for the LUT cascade emulator. The labels of the vertical axis show function names. The horizontal axis shows the percentage of the processing time for BDD for CF generation (*BDD*); LUT cascade synthesis (*cascade*); C code generation and compilation (*compile*); and memory mapping (*mapping*). *overall* denotes the average percentage of each process. Fig. 14 shows that BDD generation and compilation consume most of the CPU time. For BDD generation, optimization of variable orders for BDDs consumes most of the CPU time. Reduction of the code size, e.g., reduction of the number of cells, is effective to reduce the compilation time. Increase of cells increases the simulation setup time. The time for memory mapping increases with the number of cell inputs. The amount of memory for each cell is $2^k \cdot u$, where k is the number of inputs of a cell, and u is the number of outputs of a cell. Therefore, the number of cell inputs influences the total amount of data. In our experiment, since the average number of cell inputs is small, the memory mapping time did not influence the setup time.

VI. Conclusion and Comments

In this paper, we showed a cycle-based logic simulator using the LUT cascade emulator running on a standard PC. This method converts the circuit into LUT cascades. Then, it stores the LUT data in the memory of the LUT cascade emulator. Next, it generates the program code for the control circuit of the LUT cascade emulator. The program code is suitable for logic simulation on a standard PC due to the better memory reference patterns. Experimental results using benchmark functions show that this method is 12-64 times faster than LCC on a standard PC.

Our method converts the circuit into BDDs, where the sizes of BDDs representing the circuits are limited due to the available memory. To avoid this limitation, we partition the output functions into groups. However, when the BDD representing a single output is excessively large, our method fails to perform the emulation.

One of the future projects is to derive a partition method for the circuit and to represent the circuits by smaller BDDs.

VII. Acknowledgments

This research is partly supported by Japan Society for the Promotion of Science (JSPS), MEXT, and Kitakyushu Innovative Cluster.

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] T. Sasao, Y. Iguchi, and M. Matsuura, "LUT cascades and emulators for realizations of logic functions," *RM2005*, Tokyo, Japan, Sept. 5-6, 2005, pp.63-70.

[3] T. Sasao, M. Matsuura, and Y. Iguchi, "A cascade realization of multiple-output function for reconfigurable hardware," *IWLS-2001*, Lake Tahoe , CA, June 12-15,2001, pp.225-300.

[4] T. Sasao, and M. Matsuura, "A method to decompose multiple-output logic functions," *Proc. Design Automation Conference*, San Diego, CA, USA, June 2-6, 2004, pp.428-433.

[5] P. Ashar, and S. Malik, "Fast functional simulation using branching programs," *ICCAD'95*, Nov.1995, pp.408-412.

[6] T. Sasao, M. Kusano, and M. Matsuura, "Optimization methods in look-up table rings," *IWLS-2004*, June 2-4, 2004, Temecula, California, USA, pp.431-437.

[7] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *ICCAD'93*, pp. 42-47, 1993.

[8] S. Yang, Logic synthesis and optimization benchmark user guide version 3.0, MCNC, Jan. 1991.

[9] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient boolean function manipulation," *Proc. Design Automation Conference*, June 1991, pp.52-57.

[10] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transact. Comput.*, C-35, Aug.1986, pp.677-691.

[11] R. Murgai, F. Hirose, and M. Fujita, "Logic synthesis for a single large look-up table," *ICCD1995*, pp.415-424, Oct. 1995.

[12] P. C. McGeer, K. L. McMillan, A. Saldanha, A. L. Sangiovanni-Vincentelli, and P. Scaglia, "Fast discrete function evaluation using decision diagrams," *ICCAD'95*, pp.402-407, Nov. 1995.

[13] Altera: http://www.altera.com/

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

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

[16] P. M. Maurer, "The Inversion algorithm for digital simulation," *ICCAD'94*, pp.258-261, Nov. 1994.