Communication-Efficient Hardware Acceleration for Fast Functional Simulation

Young-Il Kim Wooseung Yang Young-Su Kwon Chong-Min Kyung

Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology
Daejeon 305-701, Korea

ABSTRACT

This paper presents new technology that accelerates system verification. Traditional methods for verifying functional designs are based on logic simulation, which becomes more time-consuming as design complexity increases. To accelerate functional simulation, hardware acceleration is used to offload calculation-intensive tasks from the software simulator. Hardware accelerated simulation dramatically reduces the simulation time. However, the communication overhead between the software simulator and hardware accelerator is becoming a new critical bottleneck. We reduce the communication overhead by exploiting burst data transfer and parallelism, which are obtained by splitting testbench and moving a part of testbench into hardware accelerator. Our experiments demonstrated that the proposed method reduces the communication overhead by a factor of about 40 compared to conventional hardware accelerated simulation while maintaining the cycle accuracy and compatibility with the original testbench.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids—*Simulation, Verification*

General Terms

Verification, Performance

Keywords

Functional verification, simulation acceleration, communication overhead

1. INTRODUCTION

With system designs now well over one million gate, the verification process has become a critical bottleneck in the design process[1]. HDL simulation is most widely used methods for verifying functional designs. Although the performance of simulation technology has been steadily improved, it is still not sufficient to keep pace with the explosion of design complexity.

There are several methods to accelerate the verification speed. First, raising the abstraction level and ignoring detailed level of design can increase the verification throughput[3][8]. Second, one can accelerate the verification speed using hardware[9]. Most of hardware emulators are based on FPGAs, which are capable of processing calculation and event-intensive tasks in parallel. Third, one can obtain speed-up by moving testbench environment into real hardware. As design complexity increases, designers are forced to create huge testbenches to find problems before committing to silicon. To cope with complex testbench, in-circuit emulation or synthesizable testbench can be applied[10][12].

Hardware accelerated simulation achieves verification speed-up by using second method. System maps some components in the software simulation into the hardware platform specifically designed to speed up certain simulation operations. Most commonly, the testbench remains running in software, while the actual design being verified is running in the hardware accelerator. However, the criteria of determining a partition between simulation and acceleration is a challenging issue. To exploit the third speed-up method, some part of testbench is moved to hardware accelerator. In [7], a simulation-time-consuming part of testbench is moved to hardware accelerator and the rest of behavioral testbench is running on software simulator. Furthermore, the entire behavioral testbench is moved to hardware accelerator[5][10]. However, these methods require that testbench follows the specific testbench description style. In case where generally described testbench is applied to this method, this can bring additional modeling efforts. In another method, emulator includes multiple emulation modules which consist of processor and FPGA, and each emulation module communicates with each other through global programmable interconnect[6]. Although FPGA is used to process not only event intensive part of design but also communication between emulation modules for low overhead communication, this method does not reduce the overhead of communication between processor and FPGA.

In this paper, we introduce a new approach to accelerate functional simulation using hardware acceleration. In conventional hardware accelerated simulation, our experiments report that more than 70% of time is consumed for

DAC 2004, June 7–11, 2004, San Diego, California, USA.

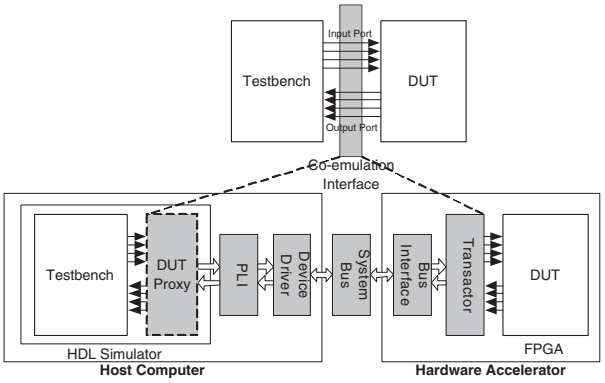

Figure 1: Acceleration system architecture

communication between software simulator and hardware accelerator. Therefore, we adopt different criteria of partition, which is optimized in communication-efficient point of view. The proposed method reduces the communication overhead without losing cycle accuracy and compatibility of the original testbench.

2. ACCELERATION SYSTEM ARCHITECTURE

High-performance verification system should incorporate both processor and FPGA. Individual processor or FPGA has a limitation in terms of performance and flexibility in simulating various types of models.

First, in view of performance, the maximum clock frequency of FPGA lags behind that of the processor implemented in contemporary ASIC. Therefore, processor with higher clock frequency executes behavioral model faster than FPGA. On the other hand, FPGA is more appropriate for executing simultaneous events and computation-intensive processes in parallel. Second, testbench is commonly created using HDL(Hardware Description Language) such as Verilog or VHDL, sometimes including C programming language, which is linked to an HDL simulator through programming language interface(PLI). This technique is used when the testbench needs to simulate more complex and higher abstract functions. FPGA is not capable of simulating the model created in C language and behavioral HDL that is not synthesizable. Therefore, processor and FPGA have mutually complementary nature for high-performance verification system.

Figure 1 shows the system architecture of conventional hardware acceleration. Testbench is performed on HDL simulator in host computer and DUT(Device Under Test) is mapped on FPGA in the hardware accelerator. As an interface between HDL simulator and FPGA, co-emulation interface is defined as aggregate of DUT proxy, PLI, device driver, system bus, system bus interface and transactor, all connected in a serial fashion. Designer-supplied testbench is interfaced with DUT proxy. When testbench applies patterns to the DUT proxy via input port, it makes input port value into a message and sends to accelerator through all the components of co-emulation interface. Likewise, the DUT proxy reads the output port value of DUT through the same path to feed them to the testbench.

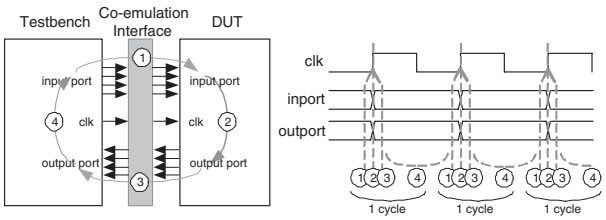

Figure 2: Conventional acceleration system performs synchronization between software simulator and hardware accelerator through co-emulation interface at every clock cycle

3. REDUCING COMMUNICATION OVERHEAD

As shown in previous section, crossing co-emulation interface is a burdensome task due to its inherited long path. However, this is inevitable to synchronize the software simulator with the hardware accelerator. In this section, we describe a detailed synchronization mechanism and propose a new process for reducing the communication overhead.

3.1 Conventional Synchronization Scheme

As shown in Figure 2, one clock cycle operation can be implemented in four steps. These four steps are as follows. When the testbench clock event occurs, step 1 is started, the co-emulation interface delivers input port value to DUT. Then, system advances one clock cycle of DUT in step 2. In this step, evaluations of DUT circuits are performed. After the result value is stabilized at the output port of DUT, in step 3, the co-emulation interface delivers the output port value to testbench. Finally, testbench checks the output results of DUT and calculates the input port value for the next clock cycle during step 4. Let us focus on that step 1 and 3 cross the co-emulation interface, thus two co-emulation interface crossing occur at every clock cycle.

The CPU time for a single clock cycle is composed of three distinct components:

$$t_{total} = t_{simulator} + t_{sync} + t_{accelerator} \tag{1}$$

where $t_{simulator}$ denotes CPU time for processing testbench, which is required for step 4. t_{sync} denotes the synchronization time for step 1 and 3. $t_{accelerator}$ denotes the emulation time for evaluating DUT circuit in step 2.
The synchronization time, t_{sync} is comprised of three components, i.e., synchronization time for input port, output port and clock port denoted as t_{inport}, $t_{outport}$ and t_{clk}, respectively, which are described as follows:

$$\begin{aligned} t_{inport} &= t_{setup} + \lceil \frac{BW_{inport}}{BW_{bus}} \rceil t_{payload} \\ t_{outport} &= t_{setup} + \lceil \frac{BW_{outport}}{BW_{bus}} \rceil t_{payload} \\ t_{clk} &= t_{setup} + t_{payload} \end{aligned} \tag{2}$$

where BW_{inport}, $BW_{outport}$, BW_{bus} denotes the bit-width of input port, output port and system bus respectively. t_{setup} is the time to set up a transaction in co-emulation interface and comprised of three terms, t_{PLI_setup}, t_{driver_setup} and t_{bus_setup}:

$$t_{setup} = t_{PLI\_setup} + t_{driver\_setup} + t_{bus\_setup} \tag{3}$$

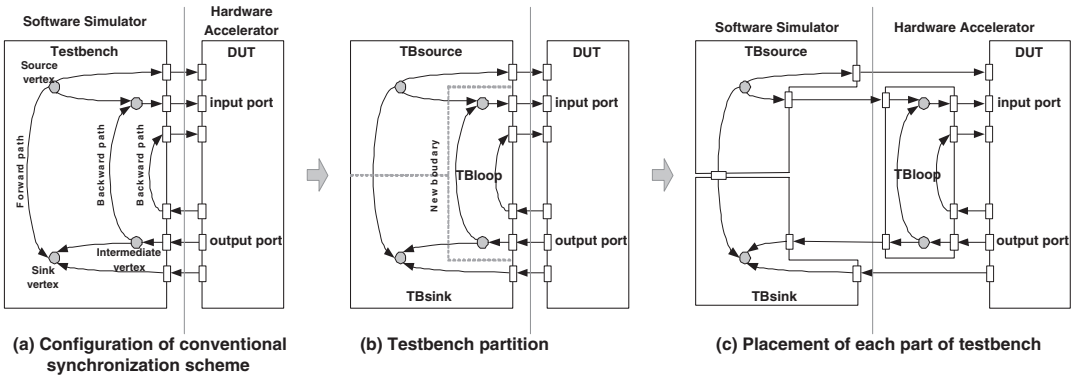

Figure 3: Procedure of the split and placement of testbench

$t_{payload}$ is the time required to send one additional word data through the co-emulation interface and is defined as follows:

$$t_{payload} = max(t_{PLI\_payload}, t_{driver\_payload}, t_{bus\_payload}) \quad (4)$$

For example, in order to make one PCI transaction, arbitration process is required to use the bus. Once bus is acquired, a number of words can be transmitted in burst data transfer. In this case, t_{bus_setup} is associated with bus arbitration time and $t_{bus\_payload}$ is the time to transmit one additional word through the bus.
Finally, the time required for synchronization is given as

$$\begin{aligned} t_{sync} &= t_{inport} + t_{clk} + t_{outport} \\ &= 3t_{setup} + (1 + \lceil \frac{BW_{inport}}{BW_{bus}} \rceil + \lceil \frac{BW_{outport}}{BW_{bus}} \rceil) t_{payload} \end{aligned} \quad (5)$$

Note that t_{setup} is obtained by summing the setup time of all the components in the co-emulation interface. On the other hand, $t_{payload}$ is obtained by finding the maximum transmit time among all three components. It is because components of co-emulation interface are connected in a serial fashion, thus setup time integrates those of all the components. On the other hand, data transmission is performed in a pipelined manner, thus total payload time becomes that of component with the lowest bandwidth. Therefore, setup time is more critical for total synchronization time, t_{sync}.

3.2 Proposed Synchronization Scheme

From the lesson of previous section, we learned that reducing t_{setup} is essential to minimize the total simulation time. Unfortunately, the amount of data for one clock cycle synchronization is fixed without compression technique, which is out of the scope of this paper. To reduce the first term of Equation (5), we perform synchronization only once for several clock cycles.

Figure 3 shows the procedure of split and placement of testbench. The testbench structure is modeled as a directed graph, which consists of vertices and edges. The vertex denotes the RTL construct and edge denotes the relation between the RTL constructs. The source vertex produces the patterns and sink vertex consumes the patterns. For example, unsynthesizable RTL block or PLI function can be a source or a sink vertex. *Path* is here defined to be composed of one or more connected edges. There are two kinds

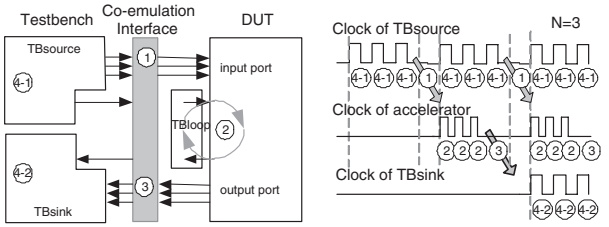

Figure 4: Proposed acceleration system performs synchronization through co-emulation interface once for a number of clock cycles

of path, which are *forward* path, *backward* path. *Forward* path is started from the source vertex and ended in sink vertex. *Backward* path is started from the output port of DUT and ended in the input port. As shown in Figure 3(b), the original testbench is split into three parts:

- TB_{source} : a part of testbench that includes the source vertices
- TB_{sink} : a part of testbench that includes the sink vertices
- TB_{loop} : a part of testbench that includes the vertices involved in *backward* path

The split procedure is performed first by finding TB_{loop}. Rest of the testbench is then split into TB_{source} and TB_{sink}. Then, TB_{loop} is mapped to the accelerator and directly attached to DUT. Therefore, changes occur in the boundary between the software simulator and hardware emulator. To cope with this, new ports are generated instead of existing ports connected to *backward* path. Finally, TB_{source} has only outgoing ports and TB_{sink} has only incoming ports. Accordingly, TB_{source} and TB_{sink} can run independently. TB_{source} can run over a number of clock cycles without waiting for data from the accelerator while TB_{sink} is executed when the output port data becomes available from the accelerator.

Figure 4 shows each clock cycle operations of the proposed acceleration system. This system is based on the same architecture depicted in Figure 1. Unlike the operation of conventional system, crossing co-emulation interface does not occur at every clock cycle. TB_{source} is executed for a number of

clock cycles but sends data through co-emulation interface only once. Let us use the notation 'N', which denotes the synchronization interval in terms of the number of clock cycles. When the TB_{source} sends synchronization data that corresponds to N clock cycles, then the accelerator performs DUT and TB_{loop} very fast by advancing N clock ticks. Finally, TB_{sink} receives the result of DUT and performs its own operation of TB_{sink} such as result checking. Because all these procedures are performed in a pipelined manner using DMA, the software simulator and accelerator can be executed in parallel, which is not possible in the conventional method. Therefore, the CPU time for a single clock cycle becomes

$$t_{total} = \max(t_{simulator}, t_{sync}, t_{accelerator}) \quad (6)$$

The time consumed by synchronizing input, output and clock port can be described as follows:

$$\begin{aligned} t_{inport} &= \frac{1}{N}t_{setup} + \lceil\frac{BW'_{inport}}{BW_{bus}}\rceil t_{payload} \\ t_{outport} &= \frac{1}{N}t_{setup} + \lceil\frac{BW'_{outport}}{BW_{bus}}\rceil t_{payload} \\ t_{clk} &= \frac{1}{N}t_{setup} + t_{payload} \end{aligned} \quad (7)$$

Since our approach just performs synchronization at the interval of N clock cycles, the first term is reduced by a factor of N but there is no effect on second term compared to the equation of the conventional method. Note that these values are normalized to one clock cycle.

Finally, the time required for synchronization can be written as

$$\begin{aligned} t_{sync} &= t_{inport} + t_{clk} + t_{outport} + t_{buffer} \\ &= \frac{3}{N}t_{setup} + (1 + \lceil\frac{BW'_{inport}}{BW_{bus}}\rceil + \lceil\frac{BW'_{outport}}{BW_{bus}}\rceil)t_{payload} \\ &\quad + t_{buffer} \end{aligned} \quad (8)$$

where BW'_{inport} and $BW'_{outport}$ denote the bit-width of ports (connected to accelerator) of TB_{source} and TB_{sink}, respectively.

t_{buffer} denotes the buffering time, which is newly introduced because the proposed method does not perform the synchronization at every cycle, the co-emulation interface should store the synchronization data for N clock cycles. Depending on the buffering method, t_{buffer} can be reduced to a negligible value compared to t_{setup}. Detailed buffering methods are discussed in the following section.

4. IMPLEMENTATION

4.1 Testbench Partition

To apply our method to a given testbench, testbench should be split according to the following steps.

4.1.1 Classifying Path

The first is to classify paths by direction into *forward* and *backward* path. A given testbench is represented as a directed graph $G = (V, E)$. We find the *backward* path first using depth-first search. From the output ports of DUT, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of edges leaving v have been explored, the algorithm backtracks to explore edges leaving the vertex from which v was discovered. If the search arrives at the input port of DUT, then the path is marked as 'backward'. Similar to the *backward* path search, *forward* path search is started from a source vertex and the path is marked as 'forward' if the search arrives at the sink vertex. Some edges may be involved in both *forward* and *backward* path.

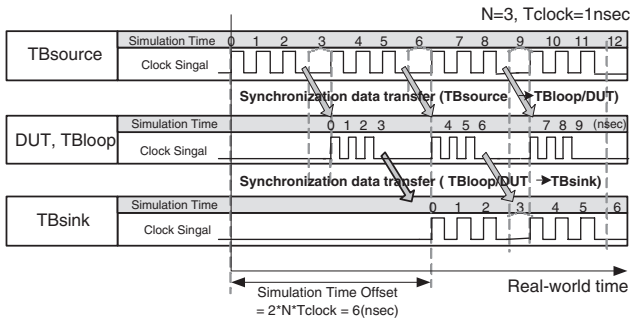

Figure 5: Simulation Time of TB_{source}, DUT/TB_{loop} and TB_{sink} in view of real-world time

4.1.2 Finding TB_{loop}

TB_{loop} should include all the *backward* paths. Edges involved in both directions can be resolved using replication. In case of strongly connected components, replication may not work. In this case, the associated *forward* path is treated as *backward* path. This may increase the total bit-width of boundary between the simulator and accelerator. However, it does not cause the critical performance degradation by the aid of burst transfer and DMA.

4.1.3 Splitting the remaining testbench into TB_{source} and TB_{sink}

After detaching TB_{loop} from the original testbench, the remaining testbench should be split into TB_{source} and TB_{sink}. As shown in Figure 3(c), *forward* path goes across the boundary between TB_{source} and TB_{sink}, therefore, new additional ports appear in TB_{source} and TB_{sink}. Data transfer from TB_{source} to TB_{sink} is performed within host computer. Although this is not critical for performance, we tried to minimize the interconnection between TB_{source} and TB_{sink}. To do this, graph partitioning is used to find a partition of the vertices of a graph minimizing the number of edges between the groups of vertices in distinct components[11].

4.2 Testbench Execution

In the simulation phase, two testbench TB_{source} and TB_{sink} are freely running on HDL simulator. These can be performed by a single process or two different processes.

Using two different processes, one simulator performs TB_{source} while the other simulator performs TB_{sink}. After starting simulations at the same time, each simulator interacts with the accelerator. This method is very straightforward, however, the execution of two simulator kernels slow down the host computer.

To overcome this problem, we use a single process for executing both TB_{source} and TB_{sink}. For example, if the original testbench is operated using a clock, TB_{source} and TB_{sink} use different clocks, which are logically the same

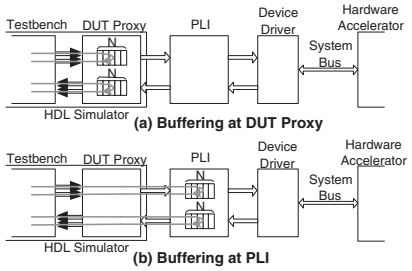

Figure 6: Buffer location in co-emulation interface

clock. This is because TB_{source} and TB_{sink} are not synchronized at every clock cycle, thus, a simulation time of TB_{source} and TB_{sink} are not mismatched in view of real-world time. The time of TB_{sink} follows that of TB_{source} with constant time offset. The time difference between TB_{source} and TB_{sink} is $2*N*(clock\ period)$. Figure 5 shows example of the simulation time progress. Let us assume that the clock period(T_{clock}) is 1nsec and we use 3 for N. In start of the simulation, TB_{source} runs while TB_{sink} is suspended for 0-6nsec of its simulation time. At 3nsec, input port data is transferred to TB_{loop} and DUT. DUT and TB_{sink} are executed and transfer the result to TB_{sink}. In 6nsec of TB_{source} time, TB_{sink} starts simulation of 0-3nsec while TB_{source} continues simulation of 6nsec-9nsec simultaneously. After the co-emulation is done, post-processing is required to match the simulation times between TB_{source} and TB_{sink} for later debugging on the waveform viewer.

4.3 Buffering Scheme

Our approach performs synchronization at intervals of N clock cycles. Therefore, co-emulation interface should include two buffers, which store the input and output synchronization data to the amount of $N * BW^{'}_{inport}$ and $N * BW^{'}_{outport}$ bits, respectively. In the middle of synchronization intervals, input port buffer is filled and output port buffer is exhausted for N clock cycles. At the time of synchronization, input port buffer is flushed to the accelerator and output port buffer is filled up.

4.3.1 Buffer Location

Figure 6 shows two different possible buffer location. The first scheme locates the buffer within DUT proxy. Because the buffer is located near the testbench, testbench can access the buffer with very low overhead. However, the buffer is implemented in HDL and this may somewhat slow down the simulation speed as N increases. The second scheme locates the buffer in PLI. Due to the distance from the testbench, this scheme has larger access time(t_{buffer}) than the first scheme. However, simulation speed is not affected by N since buffer is implemented in C programming language.

4.3.2 Buffer Implementation

We implemented buffer in five different methods. In method 1, 2, 3 and 4, buffers are located in DUT proxy. Method 5 locates buffers in PLI.

- Buffering method 1: Buffer is defined as 1-dimensional vector. The following example shows buffer declaration in Verilog:

```
reg [N*BW_inport-1:0] in_buffer;
reg [N*BW_outport-1:0] out_buffer;
```

At every clock, input and output buffer are partially accessed for $BW^{'}_{inport}$ and $BW^{'}_{outport}$ bits, respectively. The following example shows input buffer access in Verilog. There are N statements, each of which accesses statically fixed range of buffer. In this example, there is one input port of which name is *data_in* and its bit-width($BW^{'}_{inport}$) is 32-bit:

```
always @(posedge clk) begin
  case(ptr)
    0: in_buffer[31:0] = data_in;
    1: in_buffer[63:32] = data_in;
    ...
  endcase
end
```

Variable *ptr* determines which statement to be executed, which in turn select the range of buffer to access. The variable *ptr* is increased at every clock. When *ptr* arrives to value of N, synchronization is performed and *ptr* is reset to zero.

- Buffering method 2: This method is almost same to the method 1. The method 2 uses *event control* instead of *case* statement[2].

```
initial forever begin
  @(posedge clk) in_buffer[31:0] = data_in;
  @(posedge clk) in_buffer[63:32] = data_in;
  ...
end
```

- Buffering method 3: This method is also almost same to the method 1. In this case, the range of buffer is dynamically selected.

```
always @(posedge clk) begin
{in_buffer[ptr],...,in_buffer[ptr+31]} = data_in;
end
```

- Buffering method 4: Buffer is defined as 2-dimensional vector. The following example shows buffer declaration and access in Verilog:

```
reg [BW_inport-1:0] in_buffer[0:N-1];
reg [BW_outport-1:0] out_buffer[0:N-1];
always @(posedge clk) begin
  in_buffer[ptr] = data_in;
end
```

This method is very straightforward, however, PLI accessing overhead to 2-dimensional vector is bigger then 1-dimensional vector.

- Buffering method 5: Buffer is located in PLI that is described in C language. DUT proxy just transfers input port value and receives output port value from PLI at every clock cycle.

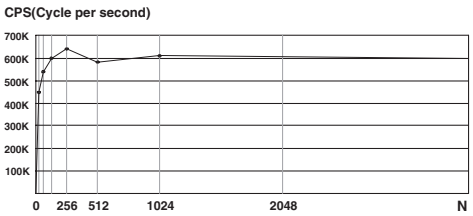

Figure 7: Simulation speed for various values of N(Synchronization interval in terms of number of clock cycles)

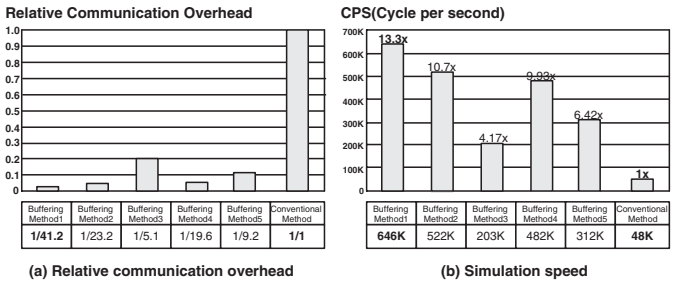

Figure 8: Proposed method reduces communication overhead by a factor of 41.2 and accelerates simulation at 646Kcps, which is 13.3 times faster than conventional method

5. EXPERIMENTAL RESULTS

This section presents the results of the co-emulation experiments. In software side, Cadence NC-Sim simulator is executed on Intel Pentium 2.8GHz processor. Hardware accelerator is implemented as a PCI card featuring 8-million gate Xilinx Virtex-II FPGA. Simulation accelerators based on FPGAs can operate at the speed 100-1000 times faster than software simulation depending on the complexity of DUT. However, we are trying to compare our approach with conventional hardware acceleration scheme rather than software simulation. To compare with the conventional method, experiments are focused on communication between the software simulator and hardware accelerator.

Figure 7 shows the simulation speed in cycles per second for various values of N. As shown in Equation (8), simulation speed increases in proportional to N before 256, however, speed is saturated after 512. This is because the first term of Equation (8) mainly influence the speed before 256, eventually, the effect of t_{buffer} becomes dominant as N increases. From the Figure 7, we experimentally found that 256 is best for N. We will use this value of N for the following experiments.

Figure 8 shows the communication overhead reduction and speed-up over conventional hardware acceleration. Commercial simulation accelerator can increase the speed of simulation by up to 100Kcps[4]. However, this figure means the maximum speed, in our experiment, we can get 48Kcps for conventional methods. On the other hand, when we use the buffering method 1, proposed accelerator reduces the communication overhead by a factor of 41.2, and performs simulation at 646K cycles per second, which is 13.3 times faster than the conventional method.

6. CONCLUSIONS

In this paper, we present new scheme that accelerates the functional simulation. For efficient acceleration of general simulation model including both behavioral and event-intensive implementation model, the acceleration system incorporates both processor and FPGA. We mainly focused on the efficiency of communication between HDL simulator and hardware accelerator. To reduce the communication overhead, we exploit burst data transfer and parallelism by eliminating a input port data dependence on output port data within testbench. It is suggested to identify a part of the testbench involved in generating next input stimulus using output results from DUT, and move it into hardware accelerator to be merged with the hardware-mapped DUT. This enables software simulator and hardware accelerator to be executed in parallel with low communication overhead. Experimental results show that the proposed method can achieve higher speed-up than the conventional hardware acceleration method. Since the proposed approach just splits the testbench, we can apply general RTL description as well as C programming language through PLI. In addition, the cycle accuracy and compatibility with the original testbench are maintained.

7. REFERENCES

[1] International technology roadmap for semiconductors. ITRS, 2001.

[2] Standard Verilog hardware description language. IEEE Computer Society, September 2001.

[3] Standard co-emulation modeling interface reference manual version 1.0. Accellera, May 2003.

[4] Platform verification white paper. Axis Corp., 2002 http://www.axiscorp.com/pdf/platform_verification.pdf.

[5] Synthesizable verification solutions. Duolog Technologies., 2002 http://www.duolog.com/verificationproducts.html.

[6] J. Bauer, M. Bershteyn, I. Kaplan, and P. Vyedin. A reconfigurable logic machine for fast event-driven simulation. Design Automation Conference, June 1998.

[7] M. Bauer, W. Ecker, R. Henftling, and A. Zinn. A method for accelerating test environments. EUROMICRO Conference, September 1999.

[8] L. Cai and D. Gajski. Transaction level modeling: An overview. CODES+ISSS, October 2003.

[9] Carlstedt-Duke and T.B.M. A solution to high performance acceleration of digital system design. Hardware Accelerators for VLSI CAD, September 1988.

[10] R. Henftling, A. Zinn, M. Bauer, M. Zambaldi, and W. Ecker. Re-use-centric architecture for a fully accelerated testbench environment. Design Automation Conference, June 2003.

[11] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:291–307, 1970.

[12] N. Kim, H. Choi, S. Lee, S. Lee, I.-C. Park, and C.-M. Kyung. Virtual chip: Making functional models work on real target systems. Design Automation Conference, June 1998.