

# A Transaction-Based Unified Simulation/Emulation Architecture for Functional Verification

Murali Kudlugi• Soha Hassoun\*  
•IKOS Systems Inc.  
{murali,selvidge,dwp}@ikos.com

Charles Selvidge• Duaine Pryor\*  
•Tufts University  
soha@eecs.tufts.edu

**Abstract** – A transaction-based layered architecture providing for 100% portability of a C-based testbench between simulation and emulation is proposed. Transaction-based communication results in performance which is commensurate with emulation without a hardware target. Testbench portability eliminates duplicated effort when combining system level simulation and emulation. An implementation based on the IKOS VStation emulator validates these architectural claims on real designs.

## 1. INTRODUCTION

Both simulation and emulation are integral parts of the functional verification of large IC designs despite the disparity of methodologies in which they are used [4, 5, 8, 6, 3, 9, 7, 10]. Typically, each requires the creation of its own stimulus driving environment (DE), an HDL testbench in the case of simulation, for example, or an in-circuit hardware test fixture for emulation. The need to create a new stimulus environment for emulation limits its adoption despite its significant model execution speed advantages.

Use of a single software-based stimulus environment for both simulation and emulation is an attractive possibility since it readily supports the migration between the two. This can combine the benefits of simulation, in the form of greater controllability and faster model turn time, with the performance advantages offered by emulation. However, to achieve significant performance advantages when using emulation, the software stimulus environment must not become a performance limitation. Most current software stimulus environments interact with the device under test (DUT) at a low level of abstraction, such as events on DUT pins within an HDL testbench. This results in both frequent model synchronization and frequent movement of small quantities of data between the stimulus environment and DUT, which in turn severely limits overall model performance, as we will show.

In contrast, the unified simulation/emulation architecture proposed in this paper overcomes the limitations of tradi-

tional simulation/emulation platforms. The main objectives of this architecture are:

- 100% portability of the DE between simulation and emulation,
- enabling hybrid representations: supporting the DE in an high level language, C, and the DUT in an HDL, and
- elimination of performance bottlenecks due to excessive DE/DUT communication.

These goals can be achieved through a transaction-based layered architecture in which only the implementations of the lower layers are changed when moving from simulation to emulation. *Transactions*, used to communicate between the DE and the DUT, encapsulate both data and synchronization information. Transactions thus require explicit, user-defined sends and receives; however, they result in fewer synchronization points between the DE and DUT. Both the DUT and DE thus may progress independently, synchronizing only when needed.

The need for efficient and systematic migration between simulation and emulation has been observed by many. Popsecu and McNamar propose using the Zycad simulation accelerator to verify the logic netlist for the emulator, thus minimizing initial debugging effort on the emulator[10]. Schnaider and Yogev have proposed using transactions for the concurrent verification of hardware and software[11]. They describe a detailed API to interface a hardware simulation engine and C code. They observe that the simulation performance is not suitable for verifying entire application C code; they state the need for interfacing the C code with an emulator, as is proposed in this paper.

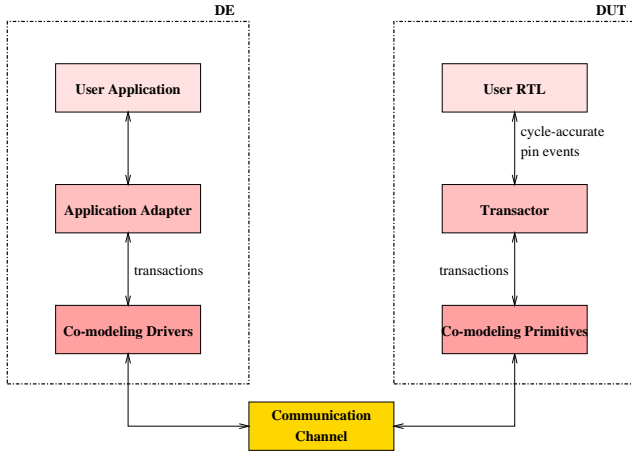
The remainder of this paper begins with an explanation of transaction-based communication, and an overview of the unified simulation/emulation architecture. Next, the architectural features common in simulation and emulation are discussed. This is followed by a description of the customization of the implementation for simulation and for emulation. Finally, experimental results and conclusions are presented.

## 2. TRANSACTION-BASED COMMUNICATION

Synchronization between different verification engines (netlist, RTL, or ISS simulators and emulators) plays a crucial role in determining the raw performance that can be achieved. Fine grain synchronization results in the entire system proceeding

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.  
Copyright 2001 ACM 1-58113-297-2/01/0006 ...\$5.00.



**Figure 1: Overview of the layered simulation/emulation architecture. The implementation of the co-modeling drivers and primitives are modified when switching from simulation to emulation. The DUT is run either using a simulator or on an emulator.**

at the rate of the slowest verification engine. This tight coupling could be implemented by cycle-based or event-based synchronization, like the one used by Bauer *et. al* for Quickturn’s event-driven behavioral simulation[2]. When synchronizing a C simulation with an HDL simulator modeling a detailed netlist, the latter becomes the bottleneck. When synchronizing a C simulation with emulation, the C code may become the bottleneck.

An alternative approach is synchronizing verification engines only when necessary via transactions. Transactions contain both data and synchronization information. They are exchanged among the engines with explicit sends and receives. Transactions allow each engine to forge ahead, performing one or more clock cycles worth of work after each synchronization point. For example, all the data for a multi-cycle bus sequence or data communication frame may be moved via a single message through one synchronization point. In contrast, with cycle-based communication, there would be one message and one synchronization point for each DUT clock cycle. Event communication results in even more messages and synchronizations.

The advantage of transaction-based synchronization v.s. cycle-based synchronization can be understood by the following simplified analysis. The initiation of one transaction for a given verification engine incurs a latency of  $L_t$  (in time units), and creates  $C_{user}$  clock cycles worth of work. One cycle-based synchronization incurs latency of  $L_{sc}$ , where  $L_{sc} \leq L_t$ , and creates only one clock cycle worth of work. If the clock has a period of  $\pi$ , then the ratio of the work done by the transaction-based engine to the cycle-based one is:

$$ratio = (C_{user} \times \pi + L_t) / (\pi + L_{sc})$$

Thus, a transaction-based verification engine will enable better performance than a cycle-based one if the latency overhead  $L_t$  is kept to a minimum and if it is possible to perform many cycles of work based on each transaction (i.e.  $C_{user}$  is large).

## 3. ARCHITECTURE

### 3.1 Overview

Figure 1 illustrates the unified, layered simulation/emulation architecture. Like traditional simulation/emulation systems, this architecture consists of a DE and a DUT connected via a communication channel. The DE, DUT and communication channel have distinct layered components.

One main component of the DE is the *User Application*. It utilizes an API provided by the *Application Adapter*, which in turn is implemented using *co-modeling Drivers*.

The DUT is comprised of the *User’s Netlist*, an *RTL Transactor*, and *co-modeling Primitives*. The transactor acts as an interface between the User’s Netlist and the underlying Primitives. The transactor transforms transactions into cycle accurate pin events.

The raw communication channel provides a mechanism for transporting data and synchronization between the DUT and DE. To communicate, the DUT and DE utilize transactions, which result in an *atomic* transfer of data and clocking information. The transaction composition and width is specified by the user. Transactions are atomic because any generated events in the DE or DUT appear at the end of the transaction.

The two top layers on both sides are common in the simulation and emulation environments. However, the implementations of the co-modeling Drivers and Primitives are different. Also the communication channels are physically different. Furthermore, all components of the DUT run on a HDL simulator in the case of simulation, while running on an emulator system when using hardware assisted verification.

### 3.2 Uncontrolled v.s. Controlled Time

To allow independent time evolution within each verification engine, or within the DE and DUT, the notion of *controlled time* modeling within the DUT is developed.

*Uncontrolled time* refers to real time (wall clock). *Controlled time* (or modeled time) refers to the time evolution as seen by the DUT. Because communication and synchronization are attributes of the modeling environment, not the model, they must not appear to consume modeled time. Thus, the DUT must be controlled by a clock that is generated based on controlled time. The DUT sees clock and data on edges of the controlled clock. The time used in transportation and processing of transactions is thus invisible to the DUT. This results in a cycle-accurate execution framework for the DUT in modeled time.

### 3.3 System Operation

The user of this simulation/emulation system provides C code for the User Application, and a Netlist for simulation/emulation. The User Application may contain a complex C model of a system component, or it may contain a test environment that provides test vectors for the Netlist. The User Netlist is the RTL or gate level model to be co-verified with the C code.

To send or wait for a transaction from the DUT, the User Application utilizes specific calls provided by the Application Adapter’s API (Application Programming Interface). A standard C API has been developed which may be used across many applications, and which can be used to build a more specialized application-specific API.

Similarly, the User Netlist utilizes signals provided by the Transactor, which is the system module responsible for processing input and output transactions to the User Netlist. Transactors unpack transactions arriving from the DE and produce a sequence of cycle-level stimuli to the DUT. Transactors also pack DUT output data and status information that the User Netlist must send to the User Application. They are thus tailored for each application. They are designed to be compatible with the co-modeling Primitives. The latter are application independent. They perform low-level synchronization between the channel and the Transactors.

Transactions can be initiated by the User Application or by the Netlist. The User Application sends a transaction by calling the appropriate API routine with the proper data. The call activates the co-modeling Drivers which send the transaction across the communication channel. If the DUT's co-modeling Primitives are busy, the transaction is buffered in the channel. Once the transaction is received via the co-modeling Primitives, it is passed to the Transactor. The Transactor unpacks the data and presents it to the User Netlist. When initiated by the User Netlist, transactions undergo the reverse process. Similar channel buffering will occur if the co-modeling Drivers are busy.

This architecture can be used in one of two modes: *data streaming* and *reactive co-modeling*. In data streaming, data vectors independent of previous DUT computations are sent continuously from the DE to the DUT. This naturally best utilizes any pipeline mechanism built into the channel. In reactive co-modeling, the User C code depends on the results of a previous transaction to generate the next transaction. The User C code thus has to wait for the DUT to process transactions and to respond before the initiation of any new transactions. The channel pipelining is less effective, and the DE and DUT may be idle awaiting new transactions.

## 4. APPLICATION ADAPTER

The Application Adapter's API provides a variety of core C routines to facilitate sending and receiving transactions. Some of these routines are described. Call `useSystem()` checks the availability of the HDL simulator or emulation hardware. If no simulation license is available or the desired emulator is in use or powered off, then this call returns error. The calls `setReadWidth(i)` and `setWriteWidth(i)` set the width of transactions. However, this is only set once, and the same widths are utilized thereafter.

The `write(data)` writes the value of data to the interface. This is a blocking call and will wait until the write operation is possible. The `read(data)` reads a value from the interface. This also is a blocking call and will wait until data is available for reading. The call `done()` terminates the simulation/emulation run. These primitives are powerful. They can be used by the User Application to perform any send/receive operation of transactions.

## 5. CO-MODELING PRIMITIVES

The co-modeling Primitives are a collection of HDL components provided to the user. The Primitives provide functionality upon which the Transactors can be built. There are four primitives: a clock module, an input module, an output module, and a Dgate module. They are illustrated in Figure 2.

The clock module Primitive is a controlled clock generator providing the user the ability to control the simulation/emulation clock, thus supporting concept of controlled modeling of time evolution in the DUT. When `PositiveEdgeEnabled` is asserted the controlled clock undergoes rising transitions in conjunction with a corresponding transition on the uncontrolled clock. When `PositiveEdgeEnabled` is not asserted, the uncontrolled clock may fall but will not rise. `NegativeEdgeEnabled` has a symmetrical effect with respect to falling clock edges. The clock module also controls Dgate modules which are latches that hold DUT data stable at times when the controlled clock is inactive.

The input module Primitive presents data from the communication channel to the user's netlist. The data is sent by the User Application through the API call `write(data)`. The input Primitive contains an input data register matching the transaction width that will temporarily hold channel data until the user's netlist (through the Transactor) accessed the data. Upon the arrival of new data, the signal `NewData` is asserted. A one on `DataDone` driven by the transactor indicates that the module may overwrite the data value during the next cycle.

The output module Primitive allows the user's netlist (through the transactor) to send data to the User's Application, where it can be read using the API call `read()`. This Primitive and the input Primitive are intended to be symmetrical in operation. Thus, when the module senses `NewData`, data is read into an output data register. Once the read operation is completed, `DataDone` is asserted.

## 6. IMPLEMENTATION

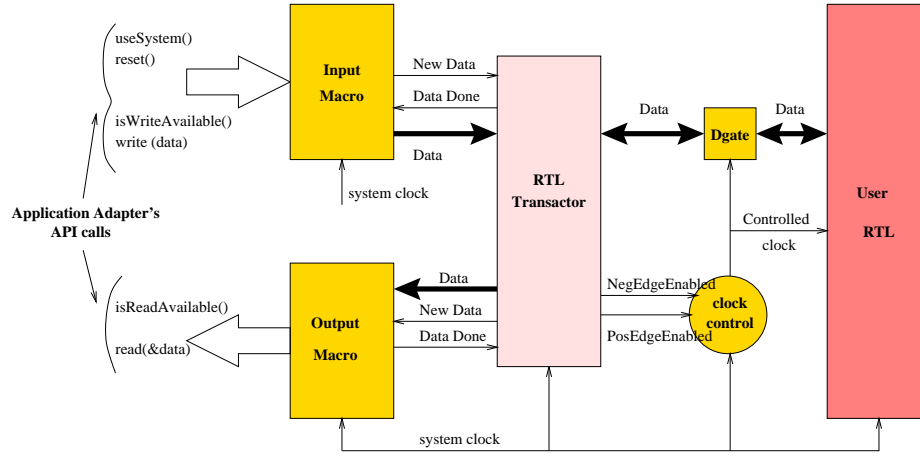
Implementing the architecture for both simulation and emulation requires customizing the low level components of the architecture in Figure 1 on both the DE and DUT sides. For simulation, the DE and DUT are realized as two processes connected through a UNIX-based, POSIX-compliant socket[12]. For emulation, the DE is implemented on a host workstation, the DUT is mapped to an emulator. The host workstation communicates with the emulator through a PCI card[1] and a specialized component, called the PCI-IB.

### 6.1 Simulation

The simulation environment consists of two executables representing the DE and DUT. The C code for the User Application is compiled together with an API library for the Application Adapter. The library contains the API functions described in Section 4. Here, the API functions contain socket-based calls to interact with the other executable.

The other executable has an HDL simulation core which runs the User RTL, Transactor code, and co-modeling Primitives. To communicate with a UNIX socket, the co-modeling Primitives (input, output, and clock) use special PLI (Programming Language Interface) calls. PLI provides a mechanism to interface Verilog programs with programs written in C language. It also provides mechanisms to access internal databases of the simulator from the C program. During execution, a provided library containing the PLI routines is dynamically linked with the simulation. Thus, the Primitives through the PLI calls maintain socket level communication between the User RTL code and User Application through a UNIX-based, POSIX-compliant sockets.

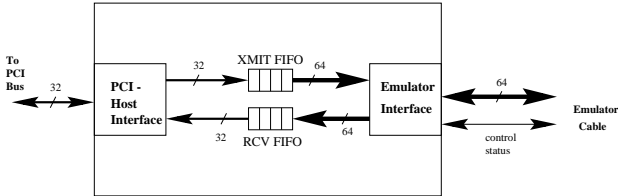
Surprisingly, the current simulation implementation proves in general to be less efficient than a pure PLI solution that



**Figure 2: Co-modeling Primitives and their interaction with the Transactor and User RTL.**

directly stimulates DUT pins without the aid of a Transactor. Some experiments, not reported here, have shown as much as a 4X slowdown. This behavior is due to the implementation of the uncontrolled versus controlled time in the case of simulation. When the HDL model is awaiting the receipt of a transaction or the processing of an outgoing transaction, it is unable to meaningfully proceed. This manifests as the advancement of uncontrolled time with controlled time being inactive. Progress is stalled until the DE model is scheduled by the underlying operating system.

A means by which unproductive controlled clock cycles can be suppressed is the yielding of control at such times to the DE model. Based on experience with the emulation implementation, this modification should result in performance that equals or exceeds that of non-transaction communication.



**Figure 3: Block diagram of the PCI-IB – the communication board between the host's PCI cable and the emulator.**

## 6.2 Emulation

For emulation, the communication channel between the User Application Side and the User RTL side is implemented via an interface board, the PCI-IB, that sits between the workstation host and the emulator. The implementation uses a Sun Workstation running Solaris 2.5.1, and a VStation-5M emulator system from IKOS. The emulator connection is made via a face-plate connect which attaches to a single emulator I/O cable. The PCI-IB is implemented primarily using an Altera Flex 10KE FPGA.

A simplified block diagram of the PCI-IB is presented in Figure 3. The PCI connection is made via the host's motherboard connector or a PCI expansion box. The PCI-IB inter-

faces to the emulator through a bidirectional 64-bit data bus and some control pins. PCI-IB implements a RCV (receive) and a XMIT (transmit) FIFO of 4 entries each. These FIFOs are also modeled in the simulation implementation in order to maintain consistency between the environments. Each FIFO can hold 4 entries of width 4K bits giving a maximum transaction size is 4K. It is only possible to read and write the entries in 32 bit chunks on the PCI side and 64 bits on the cable side, requiring multiple accesses per transmission.

To control the PCI-IB, the low level co-modeling drivers and Primitives have different implementations in the emulation environment than those used in simulation. Instead of driver code that communicates with the UNIX socket, the drivers control the PCI-IB. Similarly, the Primitives' implementation directly controls the PCI-IB emulator interface.

Whereas the controlled v.s. uncontrolled time concept currently has a negative performance consequence in a simulation environment, there is no corresponding behavior in emulation. Under emulation, since independent platforms execute the DE and DUT models, uncontrolled time advance on the DUT doesn't negatively impact execution of the DE model. In this context, the transaction-based communication delivers significant performance advantages in comparison to cycle or event-based models and communication is the performance limiting factor in DUT execution.

## 7. EXPERIMENTAL RESULTS

### 7.1 Detailed Example: A Cell Phone

The effectiveness of transaction-based verification is demonstrated using a digital cell phone design, the TI IS-54 US TDMA. More information about the design can be found at <http://www.ti.com>, search for IS-54.

The design is partitioned into three modules: a transmitter, a receiver, and a channel. The channel module models the corruption of transmitter output due to the wireless transmission environment between the base station transmitter and the cell phone receiver, as well as fading effects of a moving vehicle containing the handset receiver. The design environment applies real speech sample frames, vary the parameters of the channel model, and listens to the resulting speech sample. Three implementations (all C, C+RTL in simulation, C+gates in Emulation) of the design are eval-

	Simulation		Emulation		
	Time (secs)	CPU Utilization DUT:DE	Time (secs)	gate Count	Design speed
DT	330 s	87%:10%	11	152657	700kHz
DTR	700 s	82%:19%	13	393523	625kHz

**Table 1: Experimental results for the digital cell phone example.** The CPU utilization reports how the processor was utilized to perform both simulation and run the C code. The gate count for emulation refers to the number of primitive gates in the circuit. The number of clocks per frame describe how many DUT cycles were needed to process the design. Finally, the design speed refers to the frequency of the clock on the DUT side.

uated, for two different variations of the design. In the first, denoted by design DT Table 1, the transmitter is only modeled in RTL, while the rest of the system is modeled in C. In the second, denoted by DTR, the receiver is moved from C to RTL. A commercial synthesis tool was used to convert the C into RTL. In all cases, a total of 841 frames of speech samples were processed. The DUT performed 250 DUT clock cycles for each frame in DT, and 420 DUT clock cycles for each frame in DTR.

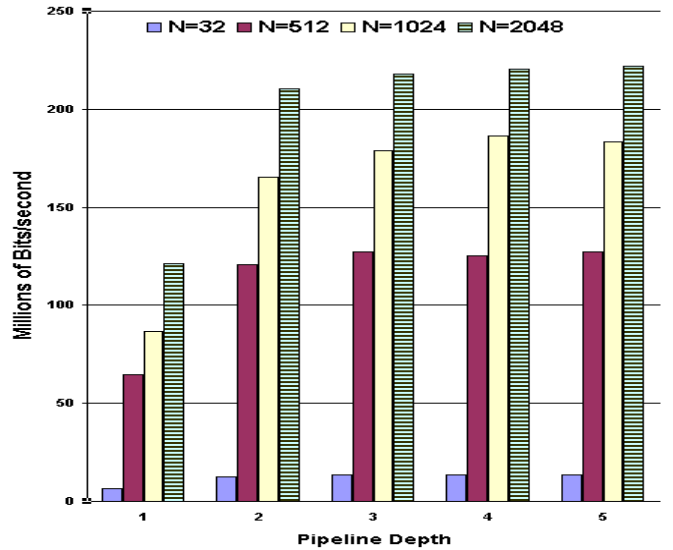
The results are reported in Table 1. Although the gate count more than doubled when more of the design was moved to Emulation, the verification time did not vary much from the all C model of 12 seconds. This demonstrates gate-level verification accuracy in the same time as running a C level model. This occurs because emulation allows more concurrency in hardware execution. Certainly, the increase in gate count adversely affects simulation.

## 7.2 Performance Benchmarks

Overall execution rate of a DUT when stimulated by a test environment is a function of the latency and bandwidth characteristics of the communication channel between stimulus and DUT as well as the total communication needs imposed by the structure of the stimulus model. Experimental results in this section demonstrate that a communication channel can be produced with capabilities consistent with emulation speeds. They also demonstrate that the precise structuring of communication between stimulus and DUT can have a dramatic impact on ultimate DUT model execution speeds.

To characterize bandwidth of the communication channel in emulation, an experiment consisting of the application of 1 million vectors to a simple DUT is performed. The DUT consists of a single, wide register. One parameter of the experiment is the width of the register, and thus width of input and output vectors, denoted  $N$ . Another is a pipeline depth  $D$ , indicating the delay tolerable in receipt of prior outputs relative to the production of new inputs by the stimulus. Results are shown in Figure 4. Bandwidth nearly attains an asymptotic level with a pipeline depth of 3 and is maximized with a maximal transaction size.

To compare the performance impact of different communication styles, an experiment is performed in which the number of communication transactions per DUT cycle is varied. Large numbers of communications per DUT cycle mimics



**Figure 4: Bandwidth, measured in MBits per second, of the communication channel as a function of transaction width,  $N$ , and pipeline depth,  $D$ .**

an event-oriented communication model. A one-to-one ratio corresponds to a cycle accurate level of communication. Many DUT cycles per communication corresponds to an abstract transaction-oriented communication style.

Results with communication occurring less frequently than once per DUT cycle are presented in Figure 5. Results with communication occurring more frequently than once per DUT cycle are presented in Figure 6. As communication becomes infrequent, overall execution asymptotically approaches the raw execution rate of the emulated DUT model, whereas when communication is very frequent, performance is completely determined by the communication channel performance and required number of communication occurrences. Note that in Figure 6, the raw execution speed of the model is effected by the total model size so all vector sizes do not asymptotically approach the same limit.

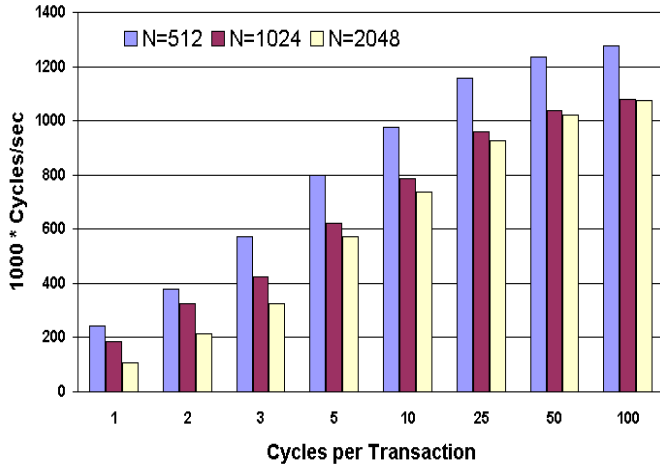
All the numbers in this section were obtained by running on an emulator running at 32 Mhz with a Sun Ultra-60 running SUNOS 5.7.

## 7.3 Additional Experiments

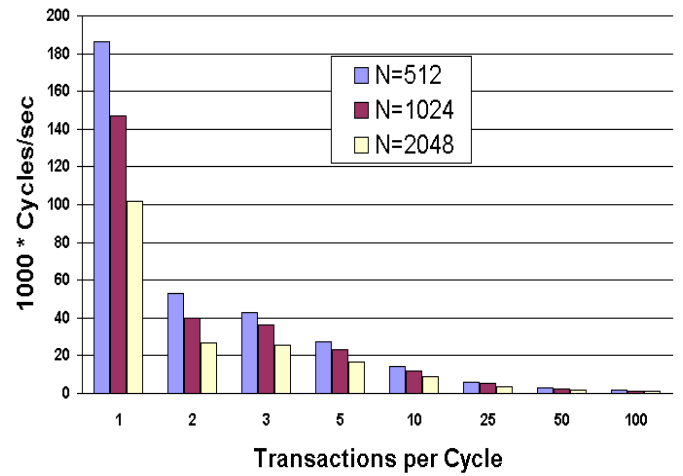
Several large industry designs with different characteristics have been validated using the proposed architecture. In each case, emulation provided considerable speedups over simulation using PLI. Table 2 reports the detailed performance of two such designs: a telecom chip that mostly operated in co-modeling reactive mode, and an IP core that was verified using test vectors in streaming mode.

## 8. CONCLUSION

The presented unified simulation/emulation architecture allows for 100% portability between simulation and emulation. It also provides a communication mechanism to concurrently exercise different verification engines (Compiled C & HDL simulators, and Compiled C & emulator). Two key enabling concepts in the implementation were using transaction-based communication and synchronization, and



**Figure 5: Results in DUT cycles per second when communication occurs less frequently than once per DUT cycle, mimicking transaction-based behavior. One cycle per transaction refers to cycle accurate performance.**



**Figure 6: Results in DUT cycles per second when communication occurs more frequently than once per DUT cycle, mimicking event-based behavior. One cycle per transaction refers to cycle accurate performance.**

	Gate Count	PLI simulation	Emulation	Speed up
Telecom IC	1.6M	20 hrs	5 minutes	240
IP Core	13.2K	5 days	23 minutes	320

**Table 2: Experimental Results for two industrial designs Telecom IC and IP Core. Emulation was two orders of magnitude faster than simulation of C and RTL through PLI.**

utilizing the controlled time concept to ensure a cycle-accurate execution framework for the DUT. Experimental data shows that performance in emulation is dramatically impacted by the style of communication between stimulus and DUT and that abstract, transaction-based communication provides maximum performance. The data also shows that transaction-based verification using C models and emulation provides cycle-based accuracy at a performance that is comparable to abstract and pure untimed C models. Furthermore, speed ups of 320X were obtained over PLI simulation.

## 9. ACKNOWLEDGMENTS

The authors wish to extend their sincere thanks to the following people: Balakrishna Nayak and Sundeep Arole for help with testcases; Dave Scott and Manish Naik for the simulation implementation discussions; John Stickly for cell phone demo and benchmarking; Varun Gupta, Jeff Evans and Andy Lindburgh for help with large examples; and Mitch Dale for useful discussions.

## 10. REFERENCES

- [1] "PCI Local Bus Specification, Revision 2.1". PCISig, 1995.
- [2] J. Bauer, M. Bershteyn, I. Kaplan, , and P. Vvedin. "A Reconfigurable Logic Machine for Fast Event-Driven Simulation". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1998.
- [3] F. Casaubielilh, A. McIssac, M. Benhamin, M. Barttley, F. Pogodalla, F. Rocheteau, M. Belhadj, J. Eggleton, G. Mas, G. Barrett, and C. Berthet. "Functional Verification Methodology of Chameleon Processor". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1996.
- [4] B. Clement, R. Hersemeule, E. Lantreibecq, B. Ramanadin, P. Coulomb, and F. Pogodalla. "Fast Prototyping: A System Design Flow Applied to a Complex System-On-Chip Multiprocessor Design". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1999.
- [5] A. Evans, A. Silburt, G. Vrckovnik, T. Brown, M. Dufresne, G. Hall, T. Ho, , and Y. Liu. "Functional Verification of Large ASICS". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1998.
- [6] G. Ganapathy, R. Narayan, G. Jorden, and D. Fernandez. "Hardware Emulation for Functional Verification for K5". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1996.
- [7] M. Kantrowitz and L. Noack. "I'm Done Simulating: Now What? Verification Coverage Analysis and Correctness Checking of the DECchip21164 Alpha Microprocessor". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1996.
- [8] N. Kim, H. Choi, S. Lee, S. Lee, I.-C. Park, and C.-M. Kyun. "Virtual Chip: Making Functional Models Work On Real Target Systems". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1998.
- [9] J. Monaco, D. Holloway, and R. Raina. "Functional Verification Methodology for the PowerPC 604 Microprocessor". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1996.
- [10] V. Popescu and B. McNamara. "Innovative Verification Strategy Reduces Design Cycle Time for High-End Sparc Processor". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1996.
- [11] B. Schnaider and E. Yogev. "Software Development in a Hardware Simulation Environment". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1996.
- [12] R. Stevens. "UNIX Network Programming, Netowkring APIs: Sockets and XTI", volume 1. Prentice Hall, 2 edition, 1997.