A Transaction-Based Unified Architecture for Simulation and Emulation

Soha Hassoun, *Senior Member, IEEE*, Murali Kudlugi, Duaine Pryor, and Charles Selvidge

***Abstract*—The availability of millions of transistors on a single chip has allowed the creation of complex on-chip systems. The functional verification of such systems has become a challenge. Simulation run times are increasing, and emulation is now a necessity. Creating separate verification environments for simulation and emulation slows the design cycle and it requires additional human efforts. This paper describes a layered architecture suitable for both simulation and emulation. The architecture uses *transactions* for communication and synchronization between the driving environment (DE) and the device under test (DUT). Transactions provide synchronization only as needed and cycle and event-based synchronization common in emulators. The result is more efficient development of the DE and 100% portability when moving from simulation to emulation. We give an overview of our layered architecture and describe its implementation. Our results show that, by using emulation, the Register–Transfer level (RTL) implementation of an industrial design can be verified in the same amount of time it takes to run a C-based simulation. We also show two orders of magnitude speeds up over simulations of C and RTL through a programming language interface.**

***Index Terms*—C, co-simulation, emulation, hardware description language (HDL), Register–Transfer level (RTL), simulation, transactions, validation, verification.**

I. Introduction

SHRINKING process technologies have allowed building million-transistor complex designs containing multiple programmable IP cores on a single chip. The verification of such systems at many levels is a challenge that must be overcome to meet time-to-market constraints. The last two decades have witnessed phenomenal verification advances and the emergence of separate strategies and tools to verify the design at the functional, timing, power, and performance levels.

A core verification technique is the construction of models depicting the design and then simulating them. Key factors that drive simulation efficiency are the abstraction level of the model (device versus gate versus Register–Transfer level (RTL) versus instruction), and the way events are propagated and evaluated. The emergence of hardware description languages (HDLs) such as Verilog and VHDL raised the abstraction level and allowed efficient event-based RTL simulation. With complicated IC designs, the cosimulation of heterogeneous system components (e.g., RTL netlists and C-based models) became necessary. Programming language interfaces (PLIs) for Verilog and VHDL have been used to bridge simulation engines. However, the synchronization between such engines plays a crucial role in determining the achievable raw performance. Finer grain synchronization results in the *entire* system proceeding at the rate of the slowest verification engine. For example, consider using cycle-based synchronization when co-verifying using two simulators. The performance becomes limited due to the cycle-based interaction between the two. It is possible that a pure HDL simulation can run faster than one using a PLI. However, during a design cycle, the human effort saved justifies the slower performance, especially when prototyping or emulation will follow the simulation.

A key advance in co-simulation is utilizing *transactions* to communicate between simulation engines. Transactions bundle several pin-accurate signals into a single atomic event, minimizing communication latency and maximizing communication bandwidth. They contain both data and synchronization information. Transactions are exchanged among the engines with explicit sends and receives. They 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 multicycle 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 clock cycle. Event-based communication results in *even* more messages and synchronization points.

The advantage of transaction-based synchronization versus 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 $L_{\text{sc}} \leq L_t$, and creates only one clock cycle worth of work. If the clock has a period of π, then the ratio of the work done by the transaction-based engine to the cycle-based one is

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

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).

Zaiq Technologies, founded as ASIC Alliance in 1996, introduced transactions to overcome the simulation communication bottleneck between a C-based test environment and an HDL netlist of a SoC-under-test (SUT) [14]. The two simulation environments are loosely coupled using *transactors*. *Transactors* are

Manuscript received August 21, 2002; revised July 23, 2003.

S. Hassoun is with the Department of Computer Science, Tufts University, Medford, MA 02155 USA (e-mail: soha@cs.tufts.edu).

M. Kudlugi, D. Pryor, and C. Selvidge are with the Emulation Division, Mentor Graphics Corporation, Waltham, MA 02451 USA.

Digital Object Identifier 10.1109/TVLSI.2004.840763

API interfaces that pack/unpack the signal bundling, if needed. Several types of transactors are used including: generator transactors that provide input data to the SUT and introduce errors for debugging purposes as needed; collector transactors that collect SUT output data and detect and report signaling violations; and a controller transactor that monitors transactor states and global time counters and that controls other transactors. Both polling and interrupt communication is allowed in the transactor implementations.

The need to describe more complex systems drove the creation and popularity of SystemC [23]. SystemC and the related methodology is capable of supporting transaction level modeling using the channel, interface, and event objects, and in a more substantial way than previously possible [10]. Instead of modeling a bus as wires with pin-accurate interfaces with connected modules, the bus protocol can be modeled in the bus and each device communicates with the bus model using a transaction-level API. This type of modeling reduces the effort required to model individual wires and also reduces the number of events in the system. The verification effort is minimized, along with improving simulation speed. The timing accuracy depends on the design tasks, and it can be set to be untimed, correct order of event, or synchronized to a clock.

The SystemC Verification Standard provides capabilities to efficiently build test environments [11]. APIs for transaction-level verification and recording are provided. A transaction-based verification methodology partitions the system into transaction-level tests, transactors and the design [3]. Tests are written at the architectural task levels. The same test can be used for RTL testing through an adapter/transactor. The tests and transactors communicate via task invocation at a level above the RTL level of abstraction. The transactors and the design communicate through RTL-level signals. Transactions thus allow writing verification tasks at the architectural level, and the transactors allow the reuse of these tests at the RTL level. Research continues to create effective methodologies and to investigate accuracy issues in transaction-level modeling [4], [7], [16].

In addition to simulation, *emulation* has become a standard way of functional verification [5], [6], [8], [9], [12], [13], [15], [20]. Compared to emulation, simulation is characterized by greater controllability and observability, and a cheaper, easier development framework. Simulation is thus typically used earlier in the design cycle, while emulation is used later to uncover bugs that require extremely long executions that cannot be achieved through simulation [8].

Until recently, when the verification methodology required the use of both emulation and simulation, each functional verification platform required the creation of its own stimulus driving environment (DE). For example, an HDL testbench is needed in the case of simulation, and an in-circuit hardware test fixture for emulation. This duplicate effort is a serious problem causing additional cost and delayed time-to-market. In addition, the migration from simulation to emulation became problematic. Large efforts were expended to minimize the effort to bring up a large design on an emulator. Popsecu and McNamar proposed using the Zycad simulation accelerator to verify the logic netlist for the emulator, thus minimizing initial debugging effort on the emulator [20]. This migration problem is more evident in large SoC designs composed of hybrid design representations.

The use of a single software-based stimulus environment for both simulation and emulation is attractive because it readily supports the migration between the two. Schnaider and Yogev have proposed using transactions for the concurrent verification of hardware and software [21]. 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. Newman describes how testbenches written in C that drove an HDL simulator were adapted to run on an emulator by only modifying a small amount of code while verifying a digital TV design [17]. Zaiq technologies recently partnered with Aptix, a provider of hardware and software solutions to create complex system FPGA-based prototypes [2]. Zaiq's tools and test environment can then be used to create a transaction-based prototype. Axis Systems advocates using transaction-based communication when modeling bus exchanges between a CPU and the RTL design [1]. Axis Systems' debugger implements transaction capture and replay, thus serving as a communication medium to correlate hardware and software execution. Cadence/Quickturn advocates the use of transactions within their emulation system by compiling a transactor onto the emulator, in a way similar to the one presented in this paper [18].

This paper details the design of a unified simulation/emulation layered architecture to eliminate the time needed to readapt the software stimulus environment for emulation and to allow partial migration of a hybrid design and its stimulus environment from simulation to emulation. Only the lowest implementation layers are changed when moving from simulation to emulation. Transactions are used to minimize synchronization points between the DUT and the DE. Our system thus has the following characteristics:

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

The remainder of this paper begins with an overview of the unified simulation/emulation architecture. Next, the architectural features common in simulation and emulation are discussed. This is followed by a detailed description of simulation and emulation. We then discuss emulation performance metrics and performance estimations. Finally, experimental results and conclusions are presented.

II. Architecture

A. Overview

Fig. 1 illustrates the unified, layered simulation/emulation architecture. Like traditional simulation/emulation systems, this

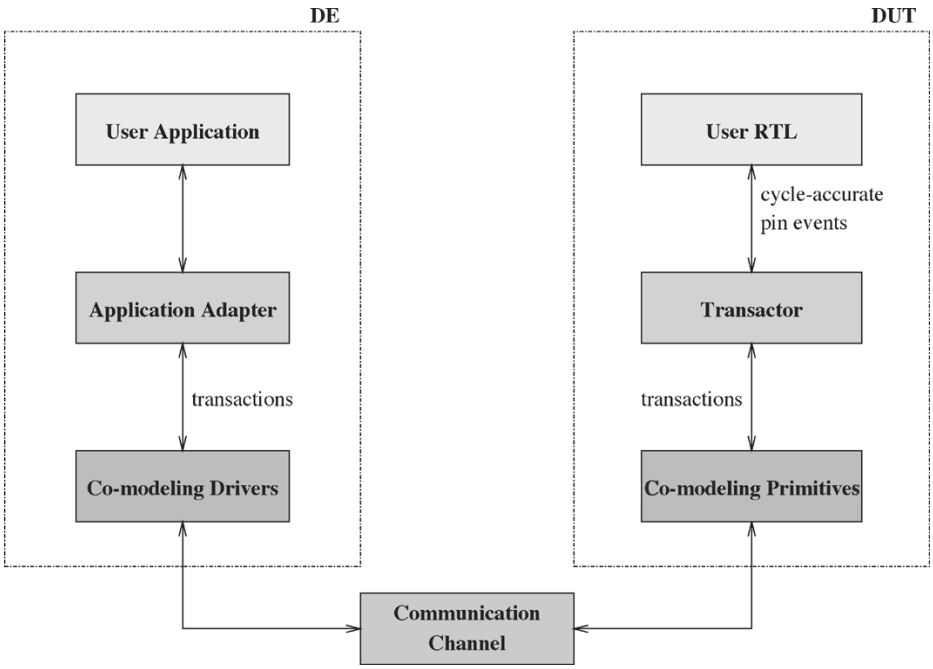

Fig. 1. Overview of the layered simulation/emulation architecture. The implementation of the comodeling drivers and primitives is modified when switching from simulation to emulation. The DUT is run using either a simulator or on an emulator.

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 an HDL simulator in the case of simulation, while running on an emulator system when using hardware-assisted verification.

B. Uncontrolled Versus 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 the clock and data on edges of the controlled clock. For example, if the workstation wants to send several transactions to the DUT *prior* to the DUT executing any data, then the next clock edge will not be generated and the DUT will not be clocked. Once all the transactions are processed and are ready to pass to the DUT, the next clock edge is generated. 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.

C. 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 application programming interface (API). 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

TABLE I
RELEVANT API CALLS PROVIDED BY THE APPLICATION ADAPTER AND UTILIZED BY THE USER APPLICATION

API call	Functionality
useSystem()	initiates the system
write(data)	blocking write of data to channel
read()	blocking read from channel
isReadAvailable()	returns true if data is available to be read
isWriteAvailable()	returns true if a write would succeed immediately
setReadWidth(i)	sets the transaction width for a read operation
getReadWidth()	gets the transaction width for a read operation
setWriteWidth(i)	sets the transaction width for a write operation
getWriteWidth()	gets the transaction width for a write operation
done()	simulation/emulation is terminated.

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. The user data is packed by the Transactor and passed through the communication channel to the co-modeling Drivers. Similar channel buffering will occur if the co-modeling Drivers are busy. Once ready, the drivers pass the transaction to the Application Adapter and then to the User Application.

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 and before the initiation of any new transactions. The channel pipelining is less effective, and the DE and DUT may be idle awaiting new transactions.

III. APPLICATION ADAPTER

The Application Adapter's API provides a variety of core C routines to facilitate sending and receiving transactions. A summary of relevant routines and related actions appears in Table I. 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, the width is set only 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. All of these primitives are powerful. They can be used by the User Application to perform any send/receive operation of transactions.

IV. 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 Fig. 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.

V. IMPLEMENTATION

Implementing the architecture for both simulation and emulation requires customizing the low-level components of the architecture in Fig. 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 [22]. 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 [19] and a specialized component, called the PCI-IB.

A. 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

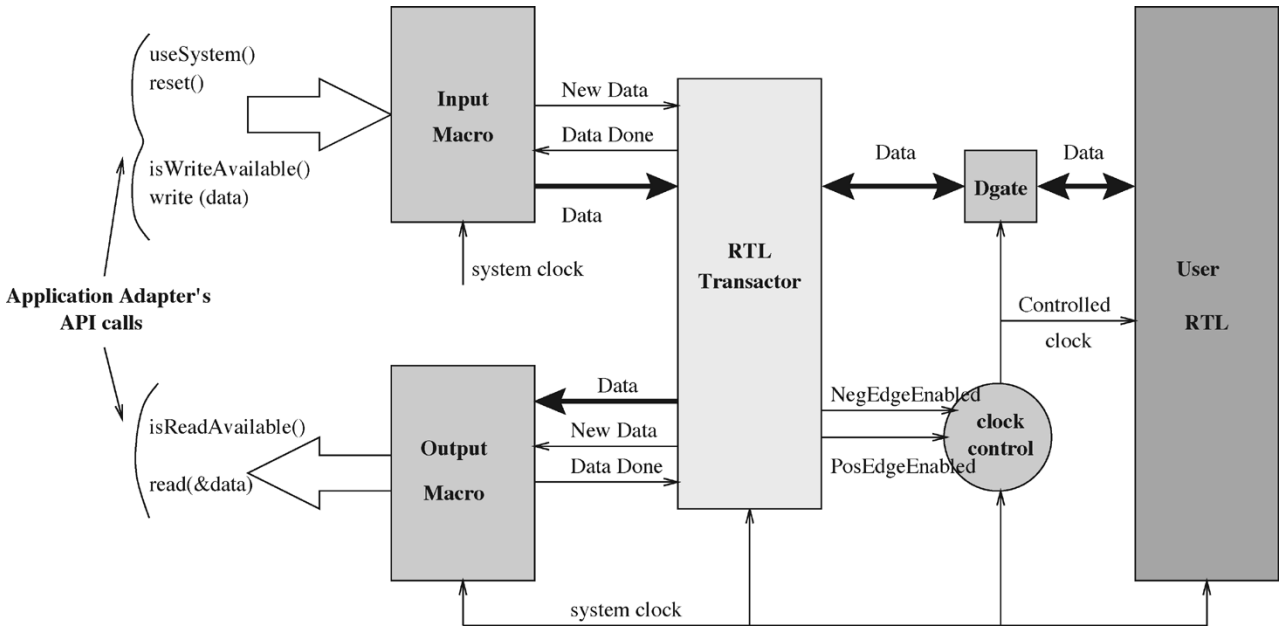

Fig. 2. Co-modeling primitives and their interaction with the transactor and user RTL.

Section III. 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 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.

The use of socket-based communication allows viewing the simulation system as a client-server model. The User Application is the client, while the User RTL is the server. This allows for several clients (user applications) that access the same RTL code. The current implementation, however, only allows one client. Moreover, it only allows the client to initiate all contact with the server, while the server acts only to serve the requests. This design decision simplified the implementation of the PLI routines where querying the client for termination was omitted.

Surprisingly, the current simulation implementation proves in general to be less efficient than a pure PLI solution that directly stimulates DUT pins without the aid of a Transactor. Some experiments, not reported here, have shown as much as a $4\times$ 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 to the DE model at such times. Based on experience with the emulation implementation, this modification should result in performance that equals or exceeds that of nontransaction communication.

B. 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 10 KE FPGA.

A simplified block diagram of the PCI-IB is presented in Fig. 3. The PCI connection is made via the host's motherboard connector or a PCI expansion box. The PCI–IB interfaces to the emulator through a bidirectional 64-b data bus and some control pins. PCI–IB implements an RCV and a XMIT FIFO of four entries each. These FIFOs are also modeled in the simulation implementation in order to maintain consistency between the environments. Each FIFO can hold four entries. Each entry is 4 Kbits wide corresponding to the maximum transaction size. It is only possible to read and write the entries in 32-b chunks on the PCI side and 64 bits on the cable side, requiring multiple accesses per transmission.

Pointers to FIFO entries for use by the software side of the PCI–IB interface are maintained explicitly by the software. FIFO pointers for the emulator side are advanced as a side effect of transmission or receipt of transaction data messages.

The width of either FIFO may be specified to be narrower than 4 Kbits in order to reduce the latency between host and emulator for a transaction. Overall latency is a monotonically increasing function of width.

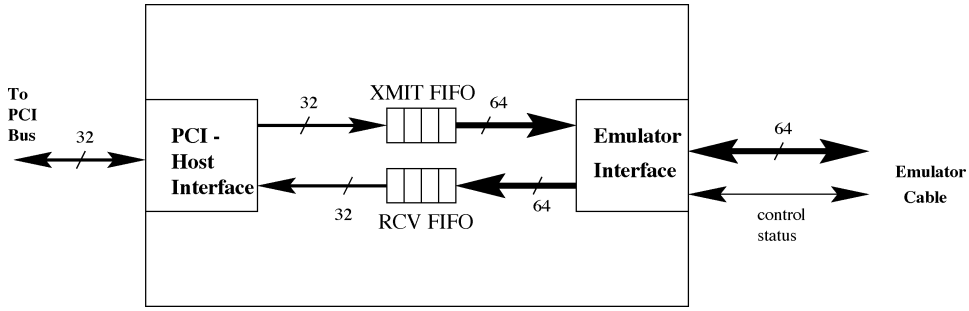

Fig. 3. Block diagram of the PCI-IB—the communication board between the host's PCI cable and the emulator.

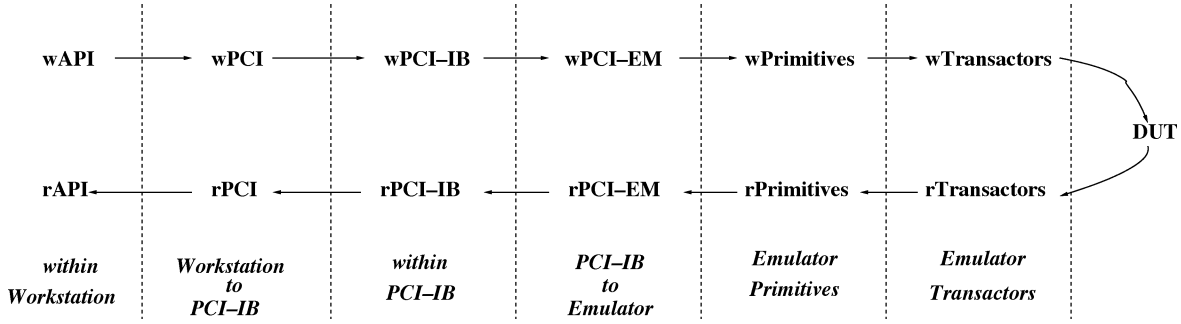

Fig. 4. Round-trip latency is the time required to perform one transaction without any added latencies within the FIFO pipeline within the PCI-IB card. It consists of the delays through each of the components shown above.

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 versus 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 does not 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.

VI. Emulation Performance Metrics and Estimation

The two performance metrics that will be used to evaluate the performance of the emulator are the **round-trip latency** and the **communication bandwidth**. In this section, we give precise definitions for these two metrics and derive quantitative estimates that we later compare against measured data.

We define the *round-trip latency* for our emulation system as the time required to perform one transaction without any added latencies of the FIFO pipeline within the PCI–IB card. This latency is depicted in Fig. 4. It consists of the following delays:

- Delays within the workstation, $w\mathrm{API} + r\mathrm{API}$, where $w\mathrm{API}$ is the time required to execute an API write to the point where a transfer from memory to the PCI-IB card is initiated, and $r\mathrm{API}$ is the time spent in the read API call. These delay estimates are given by

$$w\mathrm{API} = r\mathrm{API} = \frac{50}{M} + 3 \times \frac{N}{M}$$

where M is MIPS rating for the workstation, and N is the number of words in the transactions. The first term is the overhead of initiating the API write/read instruction, and the second term states that three clock cycles are needed to move each word through the API interface.
- Delays transferring the data from the workstation to the PCI-IB card:

$$w\mathrm{PCI} = r\mathrm{PCI} = \delta_{\mathrm{PCI}} \times p_{16} \times \left\lceil \frac{N}{16} \right\rceil$$

where δ_{PCI} is the PCI clock period, p_{16} is the number of PCI clock cycles for a 16-word read or write.
- Delays within the PCI-IB card:

$$w\mathrm{PCI-IB} = r\mathrm{PCI-IB} = \delta_{\mathrm{system}} \times c_{\mathrm{PCI-IB}}$$

where δ_{system} is the clock period of the emulator system clock, and $c_{\mathrm{PCI-IB}}$ is the number of emulator clock cycles spent on a read or a write within the PCI-IB.
- Delays transferring the data from the PCI-IB to the Emulator,

$$w\mathrm{PCI-EM} = r\mathrm{PCI-EM} = \delta_{\mathrm{system}} \times N$$

where δ_{system} is the period of the emulator system clock. Even though Fig. 3 depicts a 64-b transfer between the PCI-IB card and the emulator, only 32 of these bits are used for the data transfer while some of the other bits are control lines.

TABLE II
DATA FOR DIGITAL CELL PHONE EXAMPLE. VARIATION #1 (DT) OF THE DESIGN REFERS TO IMPLEMENTING THE TRANSMITTER ONLY IN SIMULATION/EMULATION, AND TO THE REST OF THE DESIGN IN C. VARIATION #2 (DTR) PLACES BOTH TRANSMITTER AND RECEIVER IN SIMULATION/EMULATION

	Simulation			Emulation				
	Time (secs)	Slow Down	CPU Utilization DUT:DE	Time (secs)	Slow Down	gate Count	DUT clocks/frame	Design speed
Variation #1 DT	330 s	27.5	87%:10%	11	0.91	152657	250	700kHz
Variation #2 DTR	700 s	58.3	82%:19%	13	1.08	393523	420	625kHz

Time refers to the execution time in seconds. The slow down compares the execution time to an all C system that simulates in **12 seconds**. 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 ohw many DUT cycles were needed to process the design. Finally, the design speed refers to the clock frequency of the system clock on the DUT side.

- Delays within the emulator co-modeling Primitives:

$$w\text{Primitives} = r\text{Primitives} = \delta_{\text{system}} \times c_{\text{primitives}}$$

where $c_{\text{primitives}}$ is the number of clock cycles in the primitives.

- Delays within the emulator transactors:

$$w\text{Transactor} = r\text{Transactor} = \delta_{\text{controlled\_system}} \times c_{\text{transactor}}$$

where $\delta_{\text{controlled\_system}}$ is the clock period of the controlled system clock, and $c_{\text{transactor}}$ is the number of clock cycles in the transactor.

- An application-dependent delay within the DUT.

The expressions given above are simplified estimates for the following two reasons. First, the PCI bus latencies are the most difficult to compute. They involve a software strategy to break the transactions into manageable pieces. Here, we assume all the transfers occur in 16-word bursts, regardless of the width of the transactions. Second, we assume that read and write operations for the PCI transfer, PCI-IB delays, primitives, and transactors require the same number of clock cycles. In Section VII-B, we design an experiment that allows comparing our estimates with measured delays. We only show calculations for vector sizes greater than or equal to 1024 b.

The *communication bandwidth*, CB, is the number of vectors that can be inserted and removed from a full pipeline per second. The full pipeline here assumes that the transmit and receive FIFOs, XMIT, and RCV FIFO in Fig. 3, are full. It also assumes that the DUT provides data such that the transactors do not incur any delays when a read request arrives from the co-modeling primitives.

Looking at Fig. 4, the limiting CB could possibly be the path between the workstation and the PCI-IB card

$$\frac{1}{\text{CB}} = w\text{API} + w\text{PCI} + r\text{PCI} + r\text{API}.$$

Alternatively, the path between the PCI-IB and the transactors could pose the limiting CB:

$$\frac{1}{\text{CB}} = w\text{PCI} - \text{EM} + w\text{Primitives} + w\text{Transactors} + r\text{PCI} - \text{EM} + r\text{Primitives} + r\text{Transactors}.$$

As we will see in Section VII-B, the CB is limited by the workstation to PCI-IB communication bandwidth. Our definition here of CB is justified because it enables us to do a meaningful comparison with measured data.

VII. EXPERIMENTAL RESULTS

All of the numbers in this section were obtained on an emulator running at 32 MHz with an Sun Ultra-60 running SUNOS 5.7. Additional relevant parameters are listed in Table III.

A. 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, varies the parameters of the channel model, and listens to the resulting speech sample. Three implementations of the design (all C, C+RTL in simulation, and C+gates in Emulation) are evaluated for two different variations of the design. In the first, denoted by design DT in Table II, 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 II. 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 s. 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.

B. Estimation Versus Measurements

We designed an experiment to compare our performance estimates for latency and communication bandwidth against measured emulation data. The workstation, PCI, and Station-5M emulator parameters are listed in Table III. We designed the transactor to essentially be one register of width N, the transaction width, and no DUT was used. The Application Adapter

TABLE III
WORKSTATION, PCI, AND EMULATOR PARAMETERS USED IN THE EXPERIMENT IN SECTION VII-B

	Parameter	Value
MIPS rating for host workstations	M	200 MIPS
clock period for PCI bus	δ_{PCI}	$\frac{1}{33}$ MHZ
clock period for PCI bus	δ_{system}	$\frac{1}{32}$ MHZ
clock period for PCI bus	$\delta_{controlled\_system}$	1 MHZ
number of PCI clock cycles for a 16 bit read or write	p_{16}	30 cycles
number of emulator clock cycles for a read or a write in the PCI-IB	$c_{PCI\text{-}IB}$	16 cycles
number of clock cycles in the co-modeling primitives	$c_{primitives}$	16 cycles
number of clock cycles in the transactor	$c_{transactor}$	1 cycle

TABLE IV
CALCULATED VERSUS. MEASURED ROUND-TRIP LATENCY IN MICROSECONDS, AND THE DIFFERENCE

Bits	(a) $wAPI$ + $rAPI$	(b) $wPCI$ + $rPCI$	(c) *wPCI-IB* + *rPCI-IB*	(d) *wPCI-EM* + *rPCI-EM*	(e) $wPrimitives$ + $rPrimitives$	(f) $wTransactor$ + $rTransactor$	Estimated Round Trip Latency	Measured Round Trip Latency	% Difference
1024	1.46	3.64	0.5	2	0.5	2	10.10	10.67	5.68%
2048	2.42	7.27	0.5	4	0.5	2	16.69	16.50	-1.15%
3072	3.38	10.91	0.5	6	0.5	2	23.29	23.13	-0.68%
4096	4.34	14.55	0.5	8	0.5	2	29.89	29.48	-1.36%

simply calls functions from Table I and used timers to measure transaction delays.

The estimated and measured latencies and the difference between them, all in microseconds, is shown in Table IV. The left-most column shows the number of words in a transaction. The next six columns list the estimation for the different components that contribute to the estimated latency. It is obvious that the transfer from the workstation to the PCI-IB card, $w\mathrm{PCI} + r\mathrm{PCI}$, is the dominant factor—more so as the width of the transaction is increased. The latency of the transactor is the same for all cases because of the way we designed the transactor. The last column shows the percentage difference between the estimated and measured latencies.

From the data in Table V, we can also determine the communication bandwidth, CB. The CB is limited by the workstation to PCI-IB communication bandwidth because, for each word size, the sum of columns (a) and (b) is always greater than the sum of columns (d)–(f). The larger sum in turn gives the smaller CB. As seen in Table V, the number of extracted vectors decreases with increasing vector size. However, when the bandwidth is given in millions per second, it is evident that communication bandwidth is increasing. This increase becomes smaller with increasing vector size as the transaction overheads become less dominant. Our measurements are within 10% of the estimates for the shown vector sizes.

C. Bandwidth Measurements Versus Pipeline Depth

As mentioned in Section VI, the communication bandwidth is the number of vectors that can be inserted and removed from a full pipeline. To achieve a full pipeline effect, several write requests to the emulator are issued before a read request is initiated. We define the pipeline depth D as the number of writes issued prior to a read request. We vary D for this experiment, along with the transaction width N, and we demonstrate that the maximum communication bandwidth can be achieved with little pipelining (D greater than or equal to 3). This experiment is important because it shows that the precise structuring of communication between stimulus and DUT can have a dramatic impact on ultimate DUT model execution speeds.

TABLE V
CALCULATED VERSUS. MEASURED COMMUNICATION BANDWIDTH GIVEN AS THE NUMBER OF N-BIT VECTORS INSERTED AND REMOVED FROM A FULL PIPELINE PER SECOND, AND IN MILLIONS OF BITS PER SECOND. THE LAST COLUMN IS THE DIFFERENCE BETWEEN THE CALCULATION AND MEASUREMENT

N in bits	Computed		Measured		Difference
	# of N-bit vectors	Millions of bits	# of N-bit vectors	Millions of bits	
1024	196,218	200.93	179,303	183.61	-9.4%
2048	103,170	211.29	108,305	221.81	4.7%
3072	69,983	214.99	74,962	230.28	6.6%
4096	52,951	216.89	56,380	230.93	6.1%

The transactors and application adaptor were similar to the one described in the previous section. The experiment consisted of the application of one million vectors and then measuring the number of vectors extracted on the DE side per second. Results are shown in Fig. 5. Bandwidth nearly attains an asymptotic level with a pipeline depth of 3 and is maximized with a maximal transaction size.

D. Event, Cycle, and Transaction-Based Execution Rates

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 mimic 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. The transactor and application adapter setup again is similar to the ones described in Section VII-B.

Results with communication occuring less frequently than once per DUT cycle are presented in Fig. 6. Results with communication occuring more frequently than once per DUT cycle are presented in Fig. 7. As communication becomes infrequent, overall execution asymptotically approaches the raw execution rate of the emulated DUT model, whereas when communica-

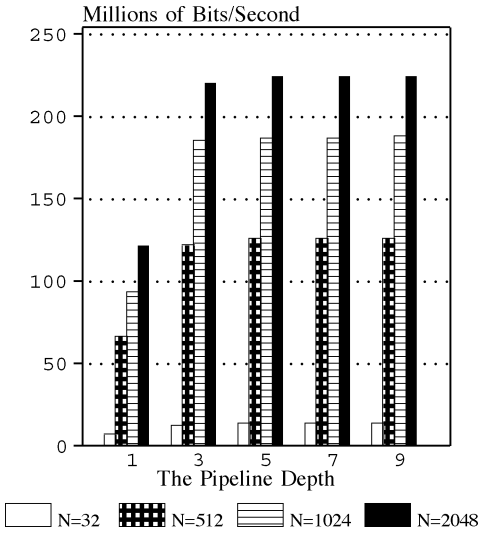

Fig. 5. Bandwidth, measured in MB per second, of the communication channel as a function of transaction width N and pipeline depth D.

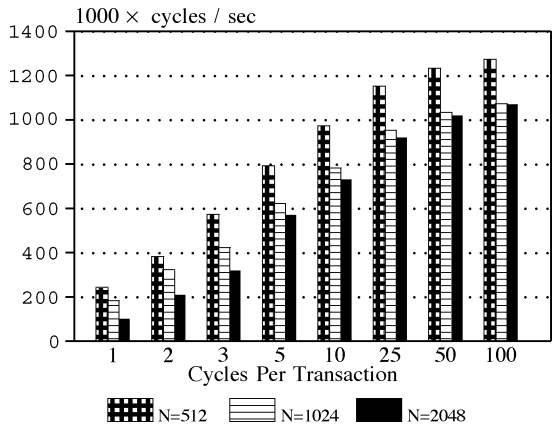

Fig. 6. Results in DUT cycles per second when communication occurs *less* frequently than once per DUT cycle, mimicking transaction-based behavior. One cycle per transaction refers to cycle accurate performance.

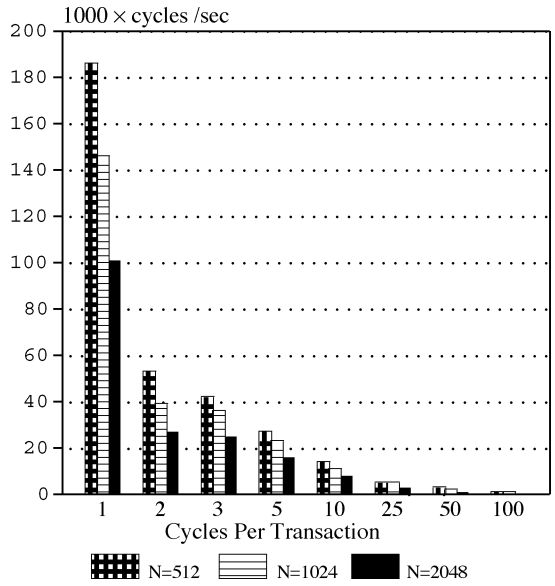

Fig. 7. Results in DUT cycles per second when communication occurs *more* frequently than once per DUT cycle, mimicking event-based behavior. One cycle per transaction refers to cycle accurate performance.

TABLE VI
EXPERIMENTAL RESULTS FOR TWO INDUSTRIAL DESIGNS: TELECOMMUNICATIONS IC AND IP CORE. EMULATION WAS TWO ORDERS OF MAGNITUDE FASTER THAN SIMULATION OF C AND RTL THROUGH PLI

	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

tion is very frequent, performance is completely determined by the communication channel performance and required number of communication occurrences. Note that, in Fig. 7, the raw execution speed of the model is affected by the total model size so all vector sizes do not asymptotically approach the same limit.

E. Additional Industrial 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 VI reports the detailed performance of two such designs: a telecommunications chip that mostly operated in co-modeling reactive mode and an IP core that was verified using test vectors in streaming mode. Both designs show significant speed ups over PLI-based simulation.

VIII. 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 and HDL simulators, and compiled C and emulator). Two key enabling concepts in the implementation were using transaction-based communication and synchronization and utilizing the controlled time concept to ensure a cycle-accurate execution framework for the DUT. Experimental data show 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 320× were obtained over PLI simulation.

ACKNOWLEDGMENT

The authors wish to extend their sincere thanks to the following people: B. Nayak and S. Arole for help with testcases; D. Scott and M. Naik for the simulation implementation discussions; J. Stickly for cell phone demo and benchmarking; V. Gupta, J. Evans, and A. Lindenburgh for help with large examples; and M. Dale for useful discussions.

REFERENCES

[1] Co-Verification Debugger Enables Hardware and Software Communication for SoC Verification. [Online] Available. http://www.axis-corp.com/products/coverification.html

[2] (2003, Feb.) Aptix and Zaiq Reseller Agreement to Improve Communication System Design Validation. [Online] Available. http://www.aptix.com/news/news.htm

[3] D. Brahme, S. Cox, J. Gallo, M. Glasser, W. Grundmann, C. Ip, W. Paulsen, J. Pierce, J. Rose, D. Shea, and K. Whiting, *The Transaction-Based Verification Methodology*. Berkeley, CA: Cadence Berkeley Labs, 2000.
[4] F. Carbognani, C. Lennard, C. Ip, A. Cochrane, and P. Bates, "Qualifying precision of abstract systemC models using the systemC verification standard," in *Proc. Design, Automation, Test in Europe*, 2002.
[5] 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. ACM/IEEE Design Automation Conf.*, 1996, pp. 421–426.
[6] 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. ACM/IEEE Design Automation Conf.*, 1999, pp. 420–424.
[7] A. Clouard, G. Mastrorocco, F. Carbognani, A. Perrin, and F. Ghenassia, "Toward bridging the precision gap between SoC transactional and cycle accurate levels," in *Proc. Design, Automation, Test in Europe Conf.*, 2002.
[8] A. Evans, A. Silburt, G. Vrckovnik, T. Brown, M. Dufresne, G. Hall, T. Ho, and Y. Liu, "Functional verification of large ASICS," in *Proc. ACM/IEEE Design Automation Conf.*, 1998, pp. 650–655.
[9] G. Ganapathy, R. Narayan, G. Jorden, and D. Fernandez, "Hardware emulation for functional verification for K5," in *Proc. ACM/IEEE Design Automation Conf.*, 1996, pp. 315–317.
[10] P. Hardee. Transaction-Level Modeling and the ConvergenSC Accelerated Transaction Based Co-Simulation Products. [Online] Available. http://www.coware.com
[11] C. Ip and S. Swan. (2003) A Tutorial Introduction on the New SystemC Verification Standard. [Online] Available. http://www.systemC.org
[12] M. Kantrowitz and L. Noack, "I'm done simulating: now what? Verification coverage analysis and correctness checking of the DEC-chip21164 alpha microprocessor," in *Proc. ACM/IEEE Design Automation Conf.*, 1996, pp. 325–330.
[13] 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. ACM/IEEE Design Automation Conf.*, 1998, pp. 170–173.
[14] A. Meyer. A Loosely Coupled C/Verilog Environment for System Level Verification. [Online] Available. http://www.zaiqtech.com
[15] J. Monaco, D. Holloway, and R. Raina, "Functional verification methodology for the powerPC 604 microprocessor," in *Proc. ACM/IEEE Design Automation Conf.*, 1996, pp. 319–324.
[16] I. Moussa, T. Grellier, and G. Nguyen, "Exploring SW performance using SoC transaction-level modeling," in *Proc. Design, Automation, and Test in Europe Conf.*, 2003, pp. 120–125.
[17] M. Newman, "Test benches in C speed verification by unifying emulation and simulation," *Integrated Syst. Design*, pp. 34–40, 1999.
[18] Cadence Application Note. (2003) Accelerated transaction based co-simulation. [Online] Available. http://www.cadence.com
[19] "PCI Local Bus Specification, Revision 2.1,", 1995.
[20] V. Popescu and B. McNamara, "Innovative verification strategy reduces design cycle time for high-end sparc processor," in *Proc. ACM/IEEE Design Automation Conf.*, 1996, pp. 311–314.
[21] B. Schnaider and E. Yogev, "Software development in a hardware simulation environment," in *Proc. ACM/IEEE Design Automation Conf.*, 1996, pp. 684–689.
[22] R. Stevens, *UNIX Network Programming, Netowkring APIs: Sockets and XTI*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1997, vol. 1.
[23] S. Swan. (2001) An Introduction to System Level Modeling in SystemC 2.0. [Online] Available. http://www.systemC.org

Soha Hassoun (SM'03) received the M.S. degree from the Massachusetts Institute of Technology, Cambridge, in 1988, and the Ph.D. degree from the University of Washington, Seattle, in 1997.

From 1988 to 1991, she was with Digital Equipment Corporation, Hudson, MA, in the Microprocessor Design Group, where she participated in several projects including the 21 064 design. Currently, she is an Associate Professor of computer science with Tufts University, Medford, MA. Her research interests include synthesis, timing and thermal analysis, and physical design.

Prof. Hassoun serves on the advisory board for ACM's Special Interest Group on Design Automation (SIGDA) and on the executive board for the International Conference on Computer-Aided Design (ICCAD). She is an Associate Editor for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN and IEEE *Design and Test Magazine*. She was the recipient of a National Science Foundation CAREER Award. In June 2000, she received the ACM/SIGDA Distinguished Service Award for creating the Ph.D. forum at the Design Automation Conference

Murali Kudlugi received the B.E. degree in computer science and technology from Bangalore University, Bangalore, India, in 1988 and the M.S. degree in computer science and engineering from the Indian Institute of Technology, Madras. in 1991. He is currently working toward the Ph.D. degree at the University of Massachusetts, Amherst.

He is an Engineering Manager with the Emulation Division of Mentor Graphics Corporation, Waltham, MA. He His research interests include system verification, HDL simulation, logic emulation, and synthesis.

Duaine Pryor received the mathematics degree (*cum laude*) from Rice University, Houston, TX, and the Ph.D. degree in mathematics from the University of California at Berkeley.

He is a Chief Technologist with the Emulation Division of Mentor Graphics Corporation, Waltham, MA. His most recent work at IKOS Systems and now Mentor Graphics has been in the area of functional verification, specifically co-modeling, the use of transaction-based techniques to accomplish system verification by combining abstract test models with an emulated design under test. In addition to his technical contributions in this area, he has chaired the Accelera SCE-MI which standardizes the co-modeling interface created by IKOS. Before coming to IKOS, his work involved scientific computing, algorithms for parallel computing, graphics, functional approximation, and pharmaceutical modeling. He is the author of numerous patents and papers in these areas.

Dr. Pryor was the recipient of the Bernard Friedman Prize in applied mathematics for his doctoral dissertation.

Charles Selvidge received the B.S., M.S., and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, in 1985, 1987, and 1992, respectively.

He is the Divisional System Architect for the Emulation Division of Mentor Graphics Corporation, Waltham, MA. His research interest is methodologies for design and verification of complex, mixed hardware/software systems.