Des Autom Embed Syst (2006) 10:105–125
DOI 10.1007/s10617-006-9586-7

# Transaction-level modeling for architectural and power analysis of PowerPC and CoreConnect-based systems

**Nagu Dhanwada · Reinaldo A. Bergamaschi · William W. Dungan · Indira Nair · Paul Gramann · William E. Dougherty · Ing-Chao Lin**

Received: 1 February 2006 / Revised: 31 March 2006 / Accepted: 3 June 2006


**Abstract** Transaction-Level models have emerged as an efficient way of modeling systems-on-chip, with acceptable simulation speed and modeling accuracy. Nevertheless, the high complexity of current architectures and bus protocols make it very challenging to develop and verify such models. This paper presents the transaction-level models developed at IBM for PowerPC and CoreConnect-based systems. These models can be simulated in a SystemC environment for functional verification and power estimation. Detailed transaction-based power models were developed. Comparisons between the simulated models and real hardware resulted in errors below 15% in timing accuracy, and below 11% in power estimation compared against gate-level power. These results demonstrate the efficiency of our transaction-level models for early analysis and design space exploration.

**Keywords** SystemC · Transaction level modeling · Architecture modeling · Power analysis · PowerPC · CoreConnect

## 1. Introduction

Complex systems-on-chip (SoC) designs today are built using pre-designed and pre-verified Intellectual Property (IP) blocks or cores. In many cases companies will also provide reference designs, or platforms, built using these IP blocks with various parameterization capabilities. Complex systems can be assembled by instantiating and connecting various cores, and/or by

N. Dhanwada (✉)· W. W. Dungan · W. E. Dougherty
IBM EDA Laboratory, Hopewell Junction, NY, USA

R. A. Bergamaschi · I. Nair
IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

P. Gramann
IBM STG, Raleigh, NC, USA

I.-C. Lin
Department of Computer Science, Pennsylvania State University, PA, USA

editing an existing platform (adding or deleting cores from it). Even if individual cores are pre-verified, a significant amount of modeling and verification is needed at the system level (hardware and software together) to ensure correct behavior from functional and performance aspects. Modeling and verification can be performed at various levels of abstraction with different degrees of accuracy. This requires the availability of simulation models for both the cores and the interconnection structures (e.g., buses).

The overall goal of system verification is to ensure that: (1) the software operates correctly from an algorithmic view, (2) the hardware operates correctly from functional and performance views, (3) the software running on the hardware operates correctly in the presence of external stimuli and constraints, and (4) the system satisfies the performance requirements. Successful design methodologies use models at different abstraction levels for the verification of the design at various stages. Previous works [1, 2] have divided up the modeling space into the following main abstraction levels: Algorithmic Level (AL), Programmers View (PV), Programmers View + Timing (PVT), Cycle Accurate (CA) and Register-Transfer Level (RTL).

The level of abstraction most suited for verifying the software on its own is the Algorithmic Level. At this level the application code is compiled and run on the host computer, with no details about the target hardware architecture. It relies on software behavioral models of the hardware peripherals (so it can interact with the environment) or simple stubs to pass data to/from peripheral devices. The AL modeling is useful for early stages of software application and algorithm development. This level may be used efficiently in systems where high-level applications have minimal interactions with the hardware, and event-driven real-time processing involving concurrency is not an issue. If there are instances where the functionality cannot be verified without tighter understanding of the interactions with the environment and their timing, an AL model will not be appropriate. Simulation at the algorithmic level is extremely fast (even faster than real time, if the host computer is faster than the embedded processor), but hardware interactions cannot be verified.

Detailed hardware verification is performed best at the Register-Transfer Level. RTL modeling relies on one or more clock signals to synchronize every computational step (i.e., the logic between two registers), thus representing a cycle accurate model of the hardware, appropriate for verifying timing critical signals and interfaces. Simulation at this level is, of course, computation intensive and inadequate for full system verification including application software running on the processor(s), which often requires millions of clock cycles. Hardware acceleration/emulation of the RTL models is somewhat adequate for system verification, but it is expensive and it can only be done after most of the system has been designed.

The middle of the modeling spectrum is occupied by the PV, PVT and CA levels. These levels are *bit-true* and register accurate, which means they provide programming interfaces (APIs) to configure the architectural registers as if the user/application code were programming the real hardware. Internal to the model, these registers may be coded in any way (e.g., using abstract data types) as long as their programming APIs make them behave like real registers to the application code. The main difference between these levels is on the timing accuracy. The PV level has no timing, but enough synchronization to enable correct functionality. The PVT level is the same as PV in functionality, but with timing added. Actions will take the correct number of clock cycles to execute, although within atomic actions not every clock tick needs to be executed. The CA level models clocking in detail, that is, all clock ticks are simulated, but contrary to RTL, the logic between clocks (or registers) is not modeled in detail. For more details on these abstraction levels, the reader is referred to [1] and [2]. These three levels (PV, PVT, CA) can be efficiently modeled by partitioning the system behavior into computation and communication parts, as it has been proposed by [3] and [4].

SystemC [5] and its transaction-level modeling style [6] are well suited for implementing this partitioning of computation and communication by allowing the separation of interface declarations from the implementation of its methods.

This paper presents the SystemC-based transaction-level models and modeling approach developed at IBM for its embedded PowerPC + CoreConnect architecture platform. These models allow the construction of full system simulation models for functional and performance verification, as well as power estimation. The approach is characterized by simple transaction definitions, scripting mechanisms for creating the system description and interconnecting modules (which do not require recompilation of SystemC code), cycle-approximate behavior and simulation speeds sufficient for running/debugging simple application code, development of device drivers, and performance evaluation. The models also include transaction-driven power functions for estimating system-level power.

The paper is organized as follows. Section 2 describes the CoreConnect architecture and the main characteristic of its main buses and components. Section 3 presents the complete details on the transaction-level models developed. Section 4 describes the approach used for power modeling and characterization within the transaction-level models. Section 5 explains the details of our execution environment, and Section 6 presents the experiments and results used for validating the approach. Section 7 presents the conclusions.

## 2. IBM CoreConnect architecture

Before describing in detail the actual transaction interfaces and methods defined for the CoreConnect architecture, it is important to explain its components and properties. The IBM CoreConnect architecture [7] consists of three buses for interconnecting cores, namely: **PLB** (Processor Local Bus [8]); **OPB** (On-Chip Peripheral Bus), and **DCR** (Device Control Register Bus). The IBM Blue Logic Core Library provides a rich set of pre-designed cores which directly interface to the CoreConnect buses, allowing designers to assemble complex SoCs in a short time.

Figure 1 illustrates how the CoreConnect architecture can be used to interconnect cores in a PowerPC-based SoC. All the buses and the devices attached to them may operate under different clocks, although the architecture dictates that the PLB, OPB, and processor clock be synchronized and exact multiples of the fastest clock.

### 2.1. Processor local bus (PLB)

The PLB is used for the fast transfer of data between high-bandwidth devices such as processor cores, external memory interfaces and DMA controllers. The PLB addresses the high performance, low latency and design flexibility issues by supporting:

- Fully synchronous, with support for both multiple masters and slaves, with various arbitration schemes
- Decoupled address, read data and write data buses with split transaction capability
- Concurrent read and write transfers yielding a maximum bus utilization of two data transfers per clock
- Address pipelining that reduces bus latency by overlapping a new write request with an ongoing write transfer and up to three read requests with an ongoing read transfer
- A sequential burst protocol allowing byte, half-word, word and double-word burst transfers. Support for 16/32/64-byte line data transfers. Architecture extendable to 256-bit data buses

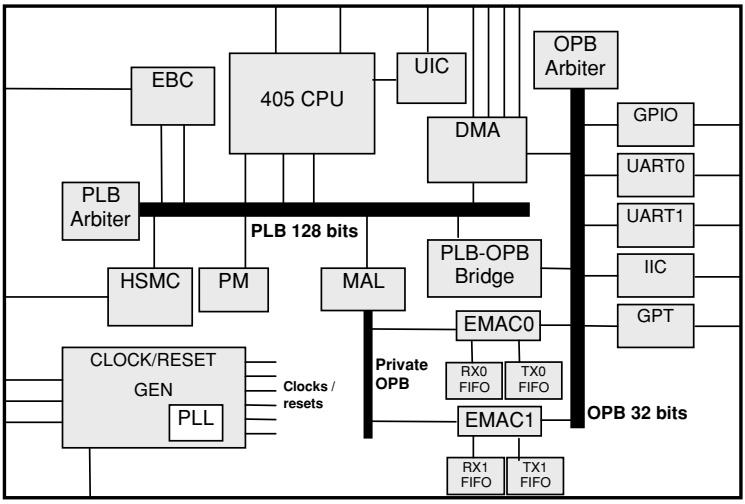


**Fig. 1** CoreConnect architecture and main components

– DMA support for buffered, fly-by, peripheral-to-memory, memory-to-peripheral, and memory-to-memory transfers
– Deadlock avoidance through slave forced PLB rearbitration
– Slave error reporting

All these characteristics need to be supported by the transaction interfaces and methods implemented by the masters, the PLB, the arbiter and the slave devices.

The PLB transaction begins with a master requesting bus ownership while transmitting a slave address and the transfer qualifiers to the bus arbiter. The arbiter selects one of the requests to service and sends the corresponding address and the qualifiers to the slaves. The slave that was configured under that address, stores it and sends an address acknowledge signal back to the arbiter. This is followed by the transfer of data between the master and the slave. When address pipelining is enabled, arbitration and address acknowledgement can be overlapped with the movement of data on the bus, thereby resulting in better bus utilization.

The decoupled address and data buses allow for address pipelining whereby the delay associated with arbitration for a new request can be overlapped with an ongoing data transfer. The PLB arbiter provides an arbitration mechanism which considers both device and request priority to determine which of several competing masters will be granted ownership of the bus in any given arbitration cycle. A locking mechanism that allows for master-driven atomic operations and a sequential burst protocol that allows for burst data transfers on the bus are also supported. In addition, the PLB arbiter provides a fixed 16-cycle bus timeout mechanism, so that a new request can be serviced in the absence of a response from the slave.

### 2.2. On-chip peripheral bus (OPB)

The OPB [7] is a secondary bus suitable for the communication with low-bandwidth devices such as serial/parallel ports, UARTs, timers and other peripherals. The OPB provides the following features:

– Fully synchronous with support for multiple masters and slaves, with separate 32-bit address and data buses
– Dynamic bus sizing to support byte, half-word and word transfers (slaves may implement different sizes data buses)
– Sequential address (burst) protocol, and bus parking for reduced-latency transfers
– Two-way bridge support between OPB and PLB
– 16-cycle fixed bus timout

2.3. Device control register bus (DCR)

The DCR [7] is a low performance bus for reading and writing status and configuration registers, so that the system does not have to waste OPB or PLB cycles for that. The DCR is fully synchronous and provides a maximum throughput of one read or write transfer every two cycles. The PowerPC processor contains special instructions for accessing the DCR bus. The DCR bus is connected to the CPU and most of the peripheral cores (DCR connections are not shown in Fig. 1 to avoid cluttering).

## 3. Modeling CoreConnect using transaction-level models in SystemC

The abstraction level adopted for these models was chosen based on a minimum set of characteristics that the system model should be able to expose and verify through simulation. This set included the following:

– Simulate real application software interacting with models for cores and the environment, for full system functional simulation and timing/performance verification potentially under real time constraints
– Verify correctness of core interconnections and communications through buses and/or other channels
– Inter-core communication should be cycle-accurate for normal operation modes, and cycle-approximate for special conditions.
– Computation (inside a core) need not be modeled on a cycle-by-cycle basis, but input-output delays should be cycle-approximate.
– Verify throughput and latency of the system running target applications
– Simulation performance should be enough to run simple software applications with a simple operating system booted on the system.

Based on these characteristics, it was decided to implement a CA model for the bus, which controls the communication between cores, and a PVT model for the computational part of most cores. In certain cases where the timing of the computation is not important, a PV model could be chosen for the core. The processor model is a particularly important one since it executes the application software. Our approach was to use an existing cycle-approximate instruction-set simulator (ISS) and *wrap* it with a SystemC interface.

The complexity of the CoreConnect buses (in particular the PLB) posed significant challenges for accurate modeling. In addition to functional and timing accuracy, another main goal was to devise transaction interfaces and methods which were simple to use and did not require detailed knowledge about the PLB or OPB protocols on the part of the user (creating the system simulation model).

### 3.1. Bus interfaces and transactions

Any communication between two cores can be decomposed into one or more steps involving an initiator and a target. In the case of a master device communicating with a slave device, the master would be the initiator and the slave would be the end target. This communication may be decomposed into 4 steps: (1) the initiator master sends a request to the target bus; (2) the initiator bus asks the arbiter for arbitration; (3) once the arbiter grants the bus to the master, the initiator bus passes the request to the target slave; and (4) the slave returns the data and/or status to the master (via the bus).

The following SystemC interfaces were defined for the CoreConnect architecture components. The initiator device class contains the port which is bound to the interface. The target device class is derived from the interface class and implements the methods supported by the interface:

(1) *plb_bus_if*: PLB Interface between a PLB master device and the PLB Bus
(2) *plb_arbiter_if*: PLB Arbiter Interface between the PLB Bus and the PLB Arbiter
(3) *plb_slave_if*: PLB Slave Interface between the PLB Bus and PLB slave devices
(4) *opb_bus_if*: OPB Interface between an OPB master device and the OPB Bus
(5) *opb_arbiter_if*: OPB Arbiter Interface between the OPB Bus and the OPB Arbiter
(6) *opb_slave_if*: OPB Slave Interface between the OPB Bus and OPB slave devices
(7) *dcr_bus_if*: DCR Interface between an DCR master device and the DCR Bus
(8) *dcr_arbiter_if*: DCR Arbiter Interface between the DCR Bus and the DCR Arbiter
(9) *dcr_slave_if*: DCR Slave Interface between the DCR Bus and DCR slave devices
(10) *intrpt_req_if*: INTERRUPT Request Interface between interrupt requester devices and the interrupt controller
(11) *intrpt_ctr_if*: INTERRUPT Controller Interface between the interrupt Controller and the interrupt servicing device (normally the CPU).

Figure 2 illustrates the relationships between these interfaces in the CoreConnect architecture. Each master connection to the bus (PLB or OPB) needs to contain specific information about that connection, such as: name, id (for priority purposes), data bus size (for dynamic bus sizing support) and address bus size (mainly checking purposes). This was implemented by using dedicated master port classes (for PLB and OPB) which register the appropriate interface class and contain dedicated data members for storing the connection specific information. Any master may have one or more connections to the bus, and each connection may have a different priority, data and address bus sizes. The bus channel can query this information from all masters and slaves (e.g., priority, address configuration, bus sizes). The bus uses it for address decoding, arbitration (through the Arbiter model) and bus sizing support.

The main methods/transactions supported by the master bus interfaces (plb_bus_if, opb_bus_if, dcr_bus_if) are: *blocking_read()/write()*, *nonblocking_read()/write()* and *direct_read()/write()*. These methods are used when a master device wants to read or write some memory location which is configured on another device connected to the architecture. Blocking read and write methods access the bus for a transaction and return only when the transaction is finished, thus forcing the master to wait. The transaction may or may not finish successfully. Non-blocking methods return immediately (without waiting for the transaction to finish) and it is the master's responsibility to check for the transaction status before using any returned transaction data. Direct methods access the bus bypassing the bus protocol thus without incurring cycle delays. They are usually used by a debugger to read/set values from/to devices without interfering with the cycle count or a running application. Additionally the

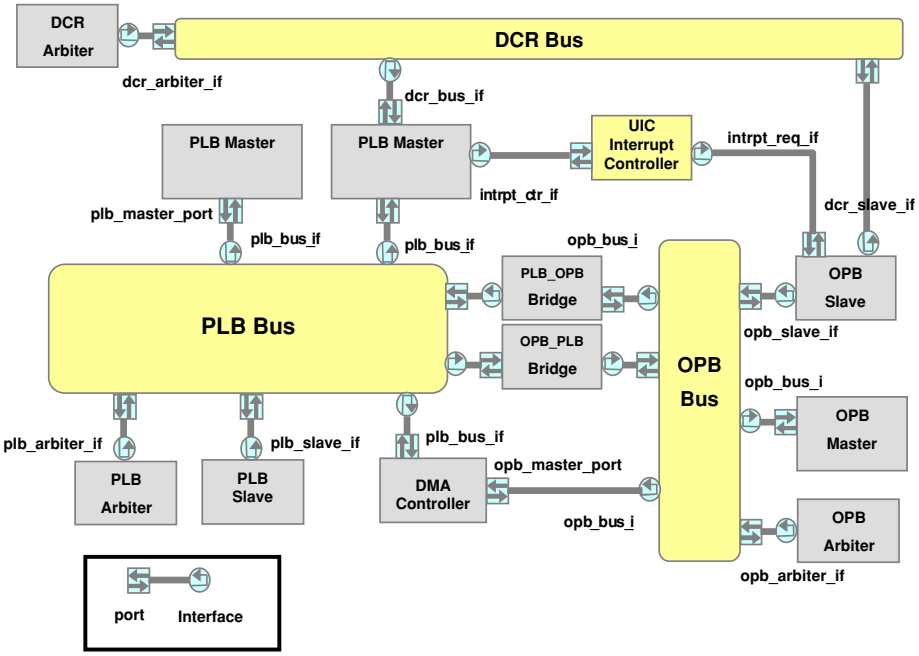


**Fig. 2** CoreConnect transaction-level modules and interfaces

master interface provides a *lock()* function to allow for atomic operations and an *abort()* function that allows a master to abort a transaction before the actual data transfer starts.

All bus slave interfaces (plb_slave_if, opb_slave_if, dcr_slave_if) support three methods: *read()*, *write()* and *acknowledge_address()*. The *read()* and *write()* methods implement the transfer of a single data unit (according to the size of the slave data bus) in one bus cycle. The *acknowledge_address()* function sets the slave status during arbitration and returns when the data address is acknowledged by the slave, or if the master aborts a transaction, or if the bus issues a timeout.

### *3.1.1. Transaction parameters*

To make the procedural interfaces for the transaction methods more extensible, we created one structure for holding all transaction data information. This structure is passed as the single parameter to all functions. Any extension that may be required later can be made by adding fields to this structure, without having to modify the function calls. As an example, the PLB_REQUEST structure contains the following fields:

```
class PLB_REQUEST {
public:
    PLB_REQ_PRIORITY_TYPE     priority;     // transaction priority
    PLB_TRANSACTION_TYPE      rw;           // read or write
    PLB_DATA_TYPE            *data;         // data buffer (array of unsigned int)
    PLB_ADDRESS_TYPE          address;      // address of first byte to be transferred
    PLB_BURST_LENGTH_TYPE     burst_length; // number of transfers of data_width
```

```
  PLB_DATA_WIDTH          data_width;    // size of data (in bytes) of each transfer
  PLB_STATUS_TYPE         bus_status;    // bus transaction status
  PLB_MASTER_STATUS_TYPE  master_status; // master transaction status
  PLB_SLAVE_STATUS_TYPE   slave_status;  // slave transaction status
  // other internal fields
};
```

The amount of data transferred per PLB (or OPB) transaction and the number of atomic transfers required may vary according to four parameters. Two static parameters that depend on the actual IP cores are the data bus sizes on the masters and slaves. The other two parameters are dynamic, i.e., may vary depending on the type of transfer requested: burst length and data width. The data buffer must be pre-allocated and its size must hold N = burst_length*data_width elements (bytes). In the case of a write operation, the data buffer will contain the bytes to be written, and for a read operation, the data buffer is filled in by the slave device and available to the calling master upon completion of the operation. The status of a transaction is maintained in this request structure in the form of three distinct transaction status fields—one each for the master, the bus and the slave. These fields are updated by the device that owns the field but can be monitored by any of the devices—thus a master could determine when a slave has acknowledged a request by monitoring the slave's status field. The status fields are also used to indicate the completion of a data transfer, either error-free or resulting in an error.

The PLB model consists of three SystemC processes that are used to model the bus behavior—a read transfer process that models the read data bus, a write transfer process that models the write data bus and an arbitration process that models the bus arbiter. The arbitration process selects a master from all the devices making bus requests and transfers the request address to the corresponding slave. In order to make this selection, the arbiter considers both request and device priorities and the depth of the read and write address pipes. The read and write transfer processes select requests that have been acknowledged by the slave and perform the actual data transfer operation. In addition, the model also includes a process that is used to determine when the bus times out, which happens after waiting for an acknowledgment from the slave for a specified number of cycles after an address has been sent to the same slave.

### 3.2. Wrapping a PowerPC instruction set simulator in SystemC

The PowerPC processor model used in this work consists of an Instruction Set Simulator (ISS) compliant with the PowerPC 405 and 440 processors [9]. The ISS is cycle-accurate for the PowerPC405 and architecturally accurate for the PowerPC440. The ISS is tightly coupled with a dedicated debugger—RISCWatch [10], which allows full access to all architectural registers as well as regular debugging capabilities on source code and assembly code. The ISS works as a standalone simulator, but also provides a set of programming interfaces for linking it with external code, such as a SystemC wrapper. Both the ISS and RISCWatch have been released as products and used in dozens of real designs.

There are two basic ways of using an ISS with a SystemC simulation environment, namely, (1) through inter-process communication (IPC) calls (e.g., sockets) handling the communication between the ISS and the SystemC modules, and (2) using a cycle-callable API for the ISS which allows the SystemC simulation to control and synchronize the ISS run on a cycle-by-cycle basis [11]. There are efficiency issues in using IPC calls for the ISS-SystemC communication, and cycle-callable APIs are not always available in a standalone ISS. If the

ISS was not originally designed to operate in reactive mode (i.e., run for one cycle upon a call from the SystemC environment), the synchronization can be more difficult. In our case, the ISS was originally designed to be the simulation driver, that is, it has its own internal clock which cannot be controlled from the outside, or in other words, there was no direct way to make the ISS clock be the same as the SystemC clock.

Instead of using IPC calls or changing the ISS code significantly to implement a cycle-callable API, we implemented a synchronization mechanism using semaphores by which the two clocks (ISS clock and SystemC clock) only toggle in sync. The ISS clock generation runs independently of the SystemC clock, but it is synchronized with the SystemC clock by locking and unlocking two semaphores, as illustrated in the following pseudo-code. There are efficiency issues in using semaphores as well, but we found this solution to be an acceptable compromise between efficiency and development effort.

| ISS clock generation code: | ISS Wrapper Thread: |
|---|---|
| ..... | |
| sem_post (&clock_semaphore1); // unlock semaphore1 | while (true) { |
| sem_wait (&clock_semaphore2); // lock semaphore2 | sem_wait (&clock_semaphore1); // lock semaphore1 |
| rising_edge_clock(); | wait (cpu_clock.posedge_event()); |
| advance_time() | sem_post (&clock_semaphore2); // unlock semaphore2 |
| ..... | } |

The ISS provides a set of APIs for interfacing with the SystemC wrapper. During execution of a program by the ISS, every time it executes an operation that accesses a memory location on the bus, it calls a function for reading or writing on that memory location. This function is a *callout* function provided by the SystemC wrapper, which implements it using the bus transactions defined in Section 3.1. The PowerPC405 and 440 may issue two bus operations at a time, one for instructions and one for data. Hence the ISS Wrapper must use non-blocking read/write calls in order to allow the ISS to issue two calls concurrently and be able to operate at its intended efficiency.

### 3.3. A simple execution flow

This section presents a simple example of the sequence of transactions called when a memory operation is executed by the ISS. In this example, illustrated in Fig. 3, a PowerPC405 processor (i.e., the ISS+Wrapper) is connected to the PLB, which is connected to the OPB via a Bridge. The OPB is also connected to a memory controller and an external memory. As shown in the example, the ISS executes operation "$*p = A + B;$", where pointer $p$ is defined as address $0\times06F00$ which is mapped to an external memory connected to a memory controller on the OPB. The ISS recognizes that the memory location is on the bus and calls the transaction *nonblocking_write()* on the data cache unit port (DCU plb_mp2). The PLB bus executes this transaction by passing a request to the arbiter through the transaction *plb_bus.plb_ap->arbitrate_request()*. Once the request is granted, the PLB queries the addresses of all slaves and determines that the PLB_OPB bridge owns the address, and it calls the transaction *write()* on the slave port *plb_sp1* connected to the bridge component. The bridge calls transaction *nonblocking_write()* on the master port connected to the OPB. The OPB determines that the address belongs to the memory controller connected to port opb_sp1 and calls the *write()* transaction on that port. Finally the memory controller passes the data

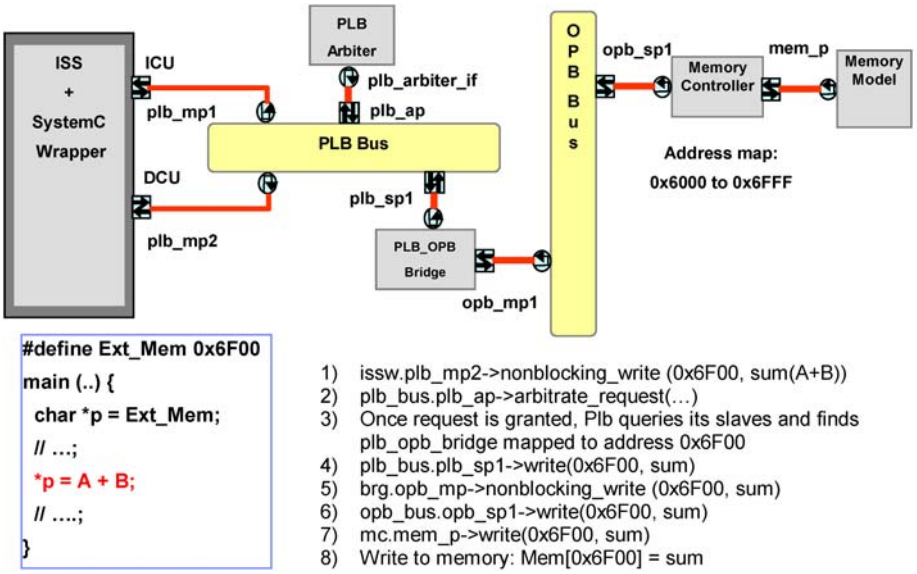


**Fig. 3** Transaction execution example

to the memory which stores it in the right address. Each of these transactions may incur zero, one or more clock cycles depending on the protocol, the size of the data and the data buses and any contention on the buses.

## 4. Power estimation using transaction-level modeling

Power and energy have emerged as important design metrics for electronic systems. Increasing design complexity is making it imperative to address power consumption at the system-level, where the benefit of performing power optimizing design changes is the greatest. The transaction-level models developed in this work were extended with power estimation capabilities to enable system designers to explore different SOC architectures, and evaluate specific design choices for both power and performance. This section describes the power modeling and estimation approach implemented as part of the PowerPC/CoreConnect SystemC transaction level simulation framework.

Power estimation, modeling and optimization can be done at different levels of abstraction. In this section we limit the discussion to system level approaches to power modeling and estimation. Instruction-based power analysis for peripheral cores was first presented in [12]. It presented a core power evaluation technique that divided the function of the cores into instructions and performed estimation using instruction level power models. In contrast, the work presented in this paper considers the case where functional transaction-level models for the cores already exist, and generates power models for those existing transactions. Integration of power models for certain components of the AMBA AHB bus into a transaction-level modeling framework was explored in [13]. In [14], a function based power estimation method was presented for embedded software executing on microprocessors. In [15] a technique for power estimation from cycle-accurate functional descriptions was presented. This approach identified the correlation between the cycle accurate functional

description and the corresponding RTL implementation for power estimation. The approach presented here is similar in that it also creates correlations, but it operates at a higher level of abstraction, namely at the transaction boundary, as opposed to cycle accurate descriptions.

An important part of this work deals with the characterization methodology which derives power values for transactions based on the detailed power computation using full gate-level simulation.

### 4.1. General power modeling approach

The overall dynamic power for an SoC can be divided into power consumed by the processor executing code and power consumed by the other cores and interconnections. In this work both parts are considered. An initial version of this work that dealt primarily with transaction level power estimation for peripherals appeared in [19]. For modeling the processor power we used the instruction/processor architectural event characterization data derived from real hardware simulations of the PowerPC processor core [16]. That data derived in [16] contained the average power for each architectural event in the processor, where an architectural event corresponds to types of instructions (e.g., load/stores, cache misses, arithmetic instructions). In our approach, as the ISS executes instructions, it calls a callback routine at the completion of each instruction. At that point, the callback routine checks the type of instruction and adds up the corresponding energy value derived in [16]. Given the total execution time of the code, the system computes the average power consumed by the processor (executing code).

The second part of the SoC power, namely, the power on the other cores and interconnections was also modeled. As described in Section 3.3 any task in a transaction-based simulation gets decomposed into a sequence of transaction calls. By associating a power value with each transaction and adding up the values, the same functional simulation can also compute the total power used by the system for the corresponding task. Most transaction-level models however, including those developed in this work, are meant to be functional models and do not necessarily map onto tasks that can be directly characterized for power. The power modeling and characterization approach is one of the unique features of the transaction level power modeling approach presented in this work.

### 4.2. Power modeling of TLMs

Transactions may be defined at different levels of granularity. A given task, such as moving data from one memory location to another, may be decomposed into fine-granularity transactions such as *send_address()*, *acknowledge_address()*, *get_data()*, etc., or implemented as a coarse-granularity transaction such as *move_data()* from one address to another. All these transactions may be characterized for power and trade-offs between accuracy and simulation speeds are applicable.

We defined a general power model organized as a Hierarchical Transaction Level Power (HTLP) tree structure, where nodes of the tree are transactions for which power is characterized. The nodes are organized in levels; and edges between nodes denote containment relationships between the transactions represented by the nodes. Figure 4 presents an example of such an HTLP tree organized into four different levels. As we move from the lower levels to higher levels, the granularity decreases, improving simulation speed at the cost of reduced accuracy. Level 0 contains the distinct phases of a transaction such as the address or data transfers. Level 1 is the individual transaction level—where each node represents a simple transaction, similar to those described in Section 3.1. The transactions in Level 1 are, for example, read, write, initialize, and burst-mode transactions. Such transactions when

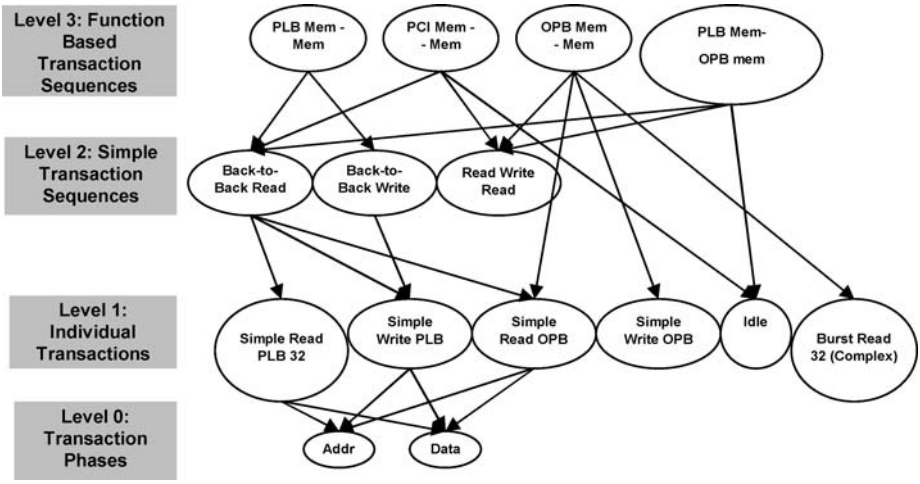


**Fig. 4** Sample HTLP-tree structure

used in simple sequences like a back to back read, or a back to back write, comprise the Level 2 nodes of this tree. Level 3 nodes represent function-based transaction sequences denoting more complex sequences of level 2 or level 1 transactions. These indicate a particular operation of the core such as a scatter-gather operation of a DMA controller, or an OPB/PLB memory-memory transfer.

The reasoning for such a hierarchical organization of the transaction-level power data can be justified using the power simulation data for a high level test, such as a DMA memory to memory transfer through the PLB. Figure 5 shows the switching activity frequency for various parts of the transfer as measured in the gate-level simulation. The switching activity is directly proportional to the power of the corresponding core. From this plot one can clearly see that the switching activity (and thus the power) fluctuates considerably during the course

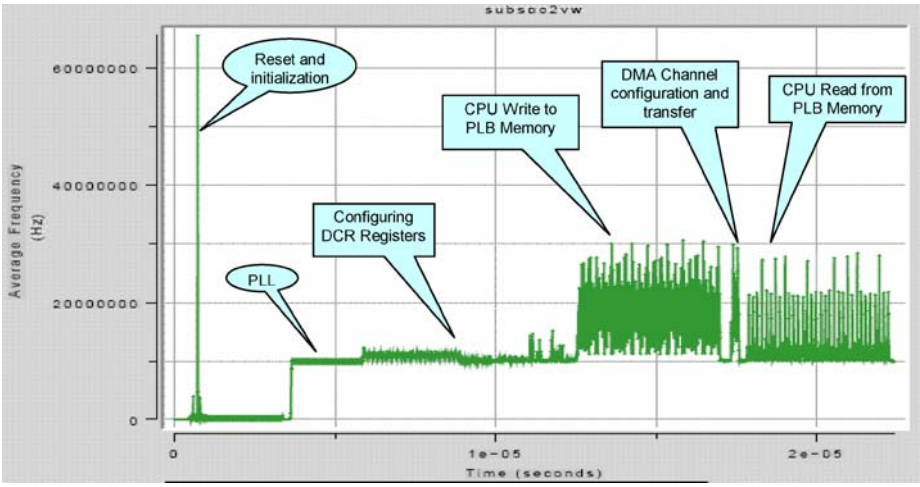


**Fig. 5** Power simulation results for DMA controlled PLB memory to PLB memory transfer

of the transfer and its main components are: (1) Configuring DCR registers, (2) Write to PLB Memory, (3) DMA operation, and (4) Read from PLB memory. The plot also shows initialization and PLL activity but these are part of the setup and not part of one specific transaction. At a coarse level, the whole transfer may be seen as one complex transaction and an average power value could be associated with it. This value, however, would not be very accurate if the transfer parameters change. The DCR register configuration would remain the same, but the transfer power would be heavily dependent on the data sizes. If however, this operation is represented as four separate transactions as indicated in the figure, then the overall accuracy would improve significantly in the case of changing parameters.

In our implementation, the transaction-level methods for each core were linked to this HTLP tree, such that, whenever a transaction method is called during simulation, the corresponding power value is obtained from the HTLP tree and added to the power being estimated for the system. This power value for each transaction may be a fixed value (as in the register configuration phase), or parameterized according to the data and mode of the transaction (i.e., data size, byte or burst mode, etc.).

### 4.3. Power characterization

The HTLP tree defines the transactions which must be associated with a power function. The accuracy of this power function is very important for the validation of the whole methodology. This section describes the transaction level power characterization methodology, which is a key component of the overall power estimation approach. The characterization process for a peripheral core has to be carried out considering the spatial context of the system in which it is present, which is why we use a CoreConnect reference SoC design containing all major cores and buses as an input to the characterization process. This process started with a fully placed and routed detailed gate-level netlist, which was then submitted to parasitics data extraction. The parasitic data was necessary for accurate gate-level power simulation.

In order to obtain power values for specific transactions (nodes in the HTLP tree) for different parameter sets we ran several gate-level power simulations using input stimuli representative of the transactions. The generation of simulation input vectors for each core and for each transaction supported by the core (and present in the HTLP tree) is extremely labor intensive and we relied on a test and verification environment called TOS [17] to help automate this task. TOS is a test and verification environment designed for driving high-level test cases through an SoC containing several different hardware cores. TOS can exercise a number of core-specific test cases to verify the connectivity and interaction among the cores on the chip. TOS translates high-level test cases into specific calls to a gate-level simulator simulating the gate-level netlist of the SoC. TOS test cases use real core functions to exercise the system functionality, such as Ethernet packet transfers and DMA transfers. Each test case is designed and written for the particular core under test/verification, and is divided into two distinct parts: the application (or task) and the device driver. TOS test cases usually find bugs related to wrong interconnections and configurations of the cores.

The first step in the characterization methodology is to write test cases (application) within the TOS environment that exercise various features of the core in the context of the system within which it is present and the transactions supported by our models, including multiple parameter settings. In a DMA controller, for example, these tests may involve software initiated memory to memory on different buses (PLB, OPB), device paced transfers, scatter gather transfers, among others; and the parameters can be: Interrupt Enable, Transfer type (Memory-Memory, Peripheral-Memory, etc.), Destination data width (Byte, Half-Word, Word), Buffer Enable, Channel priority, Source location (PLB or OPB memory space), Destination location

(PLB or OPB memory space), Prefetch enabled, Terminal count enable. In our implementation we focused on creating TOS testcases representative of Level 1 transactions, although a combination of levels and transactions could have been used. Our experiments showed that using Level 1 transactions for power modeling resulted in acceptable accuracy for the applications we were interested. These testcases are fed into TOS which triggers gate-level simulations on the SoC under test. The switching activities corresponding to each transaction on each core are derived from the simulation and used for computing the dynamic power associated with each transaction on each core. Static/leakage power is also computed using the gate-count of the cores. The process is repeated multiple times using different parameter values and the average power values for each transaction are stored in the HTLP tree.

### 4.4. System-level power estimation

To perform power estimation during SystemC simulation, we integrate the HTLP-tree-based power data into the SystemC TLM-based simulation environment. Transaction-level power model calls are directly inserted into the appropriate functions of the SystemC descriptions. During simulation, the time taken in a transaction is multiplied by the transaction power to compute the energy consumed while the core was executing the transaction. If multiple transactions in the same core are active at the same time, the joint power is computed. When the core is not active an idle power value is used for the idle cycles. The average power is the total energy divided by the total number of cycles of the simulation, and it can be reported per core and for the system as a whole.

## 5. Execution environment

In a typical SystemC environment, an *sc_main()* function is created either directly by a designer coding C++ or indirectly by a designer entering the model into a schematic capture tool. *sc_main()* serves as the entry point function which the SystemC simulator calls to begin executing the model. All of the top-level model elements are instantiated and connected via C++ code in this function, however, before this model can be simulated, a static executable containing *sc_main()* needs to be compiled and linked. Depending on the kind of modeling experiments being undertaken, it is likely that the edit-compile-simulate flow will be repeated multiple times.

The environment used for the SystemC modeling work presented in this paper does not rely on creating and compiling static SystemC executables. Instead, the SystemC library and individual SystemC models are compiled and packaged into shared libraries. Then, via the Tcl scripting language, these models can be loaded into memory, connected, and simulated in IBM's SystemC framework. Because the models are compiled into shared libraries, it is not necessary to distribute header files or source code with the modeled IP. This offers the same functionality of the typical SystemC environment with the added benefit of eliminating the compilation step for the top-level model which is assembled by the designer.

At the heart of this solution is the ability to bind functions which instantiate or manipulate SystemC objects to Tcl commands. Each shared library has an entry point function which is called when that library is loaded into IBM's SystemC framework. This entry point function exposes Tcl commands through the framework for the designer to use. For each SystemC model, there will be at least one Tcl command to instantiate it. Additional Tcl commands may also be exposed which allow further interaction with specific models.

The distinction between several SystemC types is made for the purposes of parameter checking in the Tcl commands. The recognized types correspond closely to SystemC base classes, including *sc_object*, *sc_interface*, *sc_module*, *sc_port*, and *sc_attr_base*. By generalizing around these types, it was possible to come up with a core set of Tcl commands for manipulating the SystemC models in much the same way that one can from C++. This includes commands for listing objects by type, locating modules or ports by name, connecting ports to channels, and querying object attributes.

It is possible to create the Tcl script which instantiates and connects the top-level model elements either by hand or from a schematic capture tool. Once the script is loaded into IBM's SystemC framework, the simulation can be started via the Tcl "run" command. Under the covers, the "run" command simply calls *sc_start()* with the designer's specified duration. It is also possible for the designer to advance the simulation in small time increments, modify the internal state of the cores and query various simulation metrics during simulation using pre-defined Tcl commands. Using such an environment, designers may manipulate SystemC models and run simulations without expert knowledge of SystemC or C++.

## 6. Experiments and validation

Several experiments were conducted in order to validate our transaction-level models for functional correctness, cycle accuracy (in those cores where cycle accuracy was important) and power estimation accuracy. On an individual core basis, the TLMs were simulated and compared against detailed RTL simulation. The models representing the buses (the bus model and its arbiter, for PLB and OPB) were particularly important. Several different types of data transfers were exercised and the cycle count from the TLM simulation was compared against the RTL simulation. Matching cycle counts were found in most typical bus transfers, and it was deemed acceptable to have cycle approximate behavior on certain, less frequent types of operations. For full system validation we compared our TLM simulation against a real hardware board running the same applications. Details are given in Section 6.1. For power estimation validation we compared our TLM power estimation against the same test cases running on the detailed RTL model and power computed at the gate-level.

### 6.1. An ethernet packet processing example

This example consists of a simple but realistic embedded software application running on a PowerPC405-based SoC. The application initiates and manages the flow of Ethernet packets over multiple fast Ethernet interfaces on the SoC. The same application was run on a transaction-level model for the whole SoC (including the ISS for running the code) and compared against the same application running on an embedded board containing the actual SoC hardware. The SoC used is illustrated in Fig. 1: it contains a PowerPC405, PLB, OPB, and an Ethernet subsystem represented by a MAL (Media Access Layer) core and two Ethernet controllers (EMAC0 and EMAC1) connected to their respective Transmit and Receive FIFOs. The MAL core works as a dedicated DMA engine for packet traffic between memory and the EMACs. Other peripherals are also part of the model and the hardware but were not important for the execution of this application.

The software application builds packets in memory and uses the MAL engine to transfer them from the memory controller over the PLB, through the MAL and to the EMACs. The EMACs load their TX FIFOs and transmit the packets to outside the SoC. In our simulation environment, a separate module was coded to take the transmitted packets and feed them

back directly into the Receive channel (RX FIFO), and the software application took care of receiving them and storing them back into memory via the MAL, PLB and Memory controller. This "wrap" configuration facilitates simultaneous TX and RX Ethernet traffic without the need for external Ethernet traffic models. It also permits the simulation of the entire data path (memory, bus, DMA, network interface) and software associated with the process of sending and receiving Ethernet traffic on an embedded SOC. In the actual hardware a similar external connection from the TX channel to the RX channel was used.

The application performs minimal processing of the packets. It initiates and manages the descriptor ring based packet transfers, checking for errors, counting the packets and bytes transmitted and received, keeping the packet transfers running. It also keeps track of how much CPU time is spent to do this work. The application is written as a polling loop that continuously looks for the completion of both TX and RX packets to process on each enabled interface. When the CPU is in the outer polling loops (waiting for packets), it is considered idle from a utilization point of view. The application is designed to service all of the completed packets from a single interface before moving back to the polling loop to look for more work. This results in the application handling multiple packets at once when the packets are small and frequent and handling a single packet at a time, with lots of "idle" time in the polling loop, when the packets are large and infrequent. It is important to note that in an application such as this there is a significant amount of code destined for setup and configuration of the cores. In order to run the same code in the TLM and in the hardware, the TLM must be fully bit-true compliant with the hardware.

After the application transmits a predetermined number of packets, it prints statistics representing the number of packets transmitted and received over each interface, the size of the packets, the interface utilization, and the CPU utilization during the transfers. Other SoC hardware performance statistics such as PLB accesses and utilization can be monitored during the simulation using utilities available in the TLMs. The number of interfaces exercised as well as the size and number of packets transmitted is configurable. By changing these parameters and correlating the results with those running on a hardware platform with a similar configuration, we can observe the accuracy of the modeled performance. Some discrepancies are expected as a result of feature differences between the model and the actual available hardware. The primary feature differences were in PLB configuration (TLM: 128-bit PLB4, Hardware: 64-bit PLB3) and the memory (TLM: 64-bit DDR, Hardware: 32-bit SDR).

Figure 6 shows both modeled and hardware CPU utilization for different packet sizes with two Ethernet interfaces active, both concurrently sending and receiving traffic. Figure 7 shows the network interface utilizations for the same configuration. In both the hardware and simulated results, the network interface utilization is nearly 100% in all cases. At the smallest packet sizes, the CPU is nearly fully utilized keeping the interfaces running while the CPU utilization drops to about 10% for the largest packet sizes. From this graph, it can be seen that the CPU is not the bottleneck (except for the smallest packet sizes). As the packet size increases, and fewer packets are processed, the CPU becomes underutilized (in this case, the packet processing by the CPU does not depend on the packet size) and thus free to perform other tasks in the SoC. Although it is not being plotted here, the PLB bus utilization is also low, in the order of 10%. The main bottleneck is the bandwidth of the Ethernet channels (100 Mbits/s). Even with two EMACs (each full duplex RX and TX), the CPU and the PLB bus remain underutilized, and available for other tasks.

The differences shown between the modeled and the actual hardware utilizations result from the feature differences mentioned (i.e., bus sizes and memory types) above and from generalizations or abstractions in the model. The higher PLB and memory width in the TLMs explain the slightly lower CPU utilization and higher network utilization in the TLMs. Since

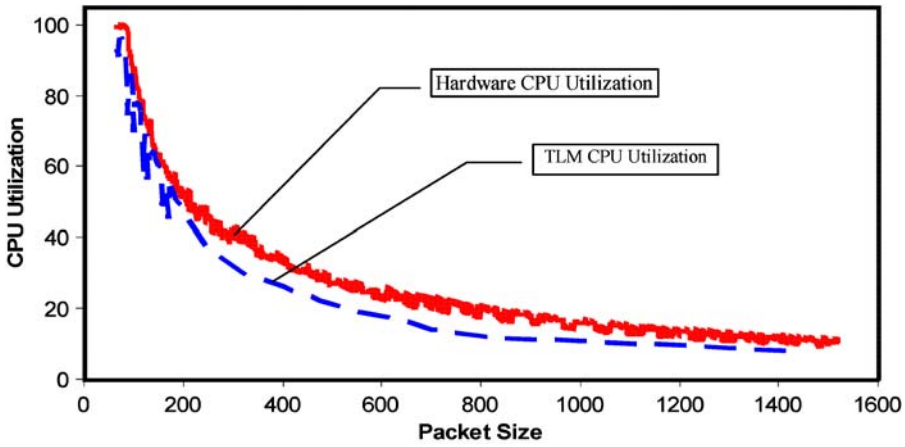


**Fig. 6** Results for CPU Utilization vs. Packet size for two Full Duplex Fast Ethernet Interfaces using TLM simulation and actual hardware measurements

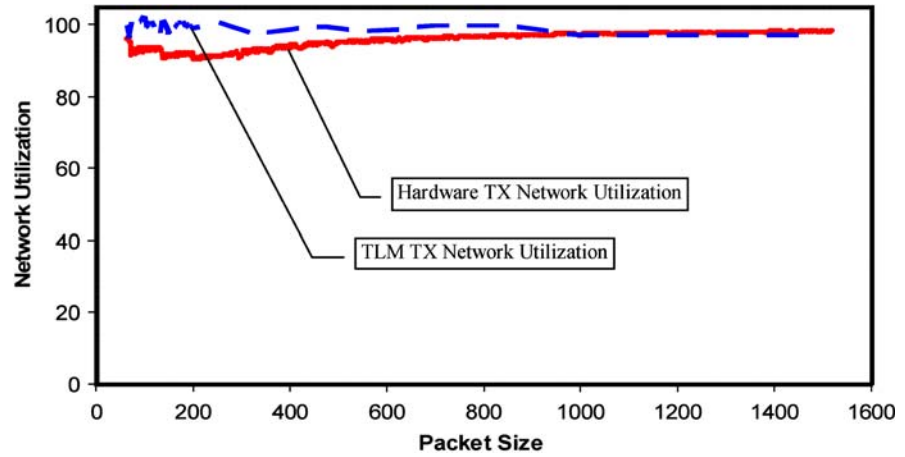


**Fig. 7** Results for Network Utilization vs. Packet size for two Full Duplex Fast Ethernet Interfaces using TLM simulation and actual hardware measurements

the PLB bus utilization in this application is low, the bus size differences did not affect results significantly. In practice, the small differences between the TLM and the hardware results demonstrate the high degree of accuracy of our transaction-level models in modeling the complex hardware and software interactions involved in managing network traffic.

Simulation performance of the SystemC models was measured on a dual processor (2.1 Ghz) Athlon machine with 3584 MB memory, running Red Hat Linux Enterprise Edition 3. With the Cache enabled for the Instruction Set Simulator the total number of instructions were 1311517, and the CPU time for the application took 24 secs, resulting in a performance of 53000 Instructions per second for this application. The total number of bus transactions generated (PLB, OPB, DCR Transactions) for this scenario with the ISS cache enabled were 150000.

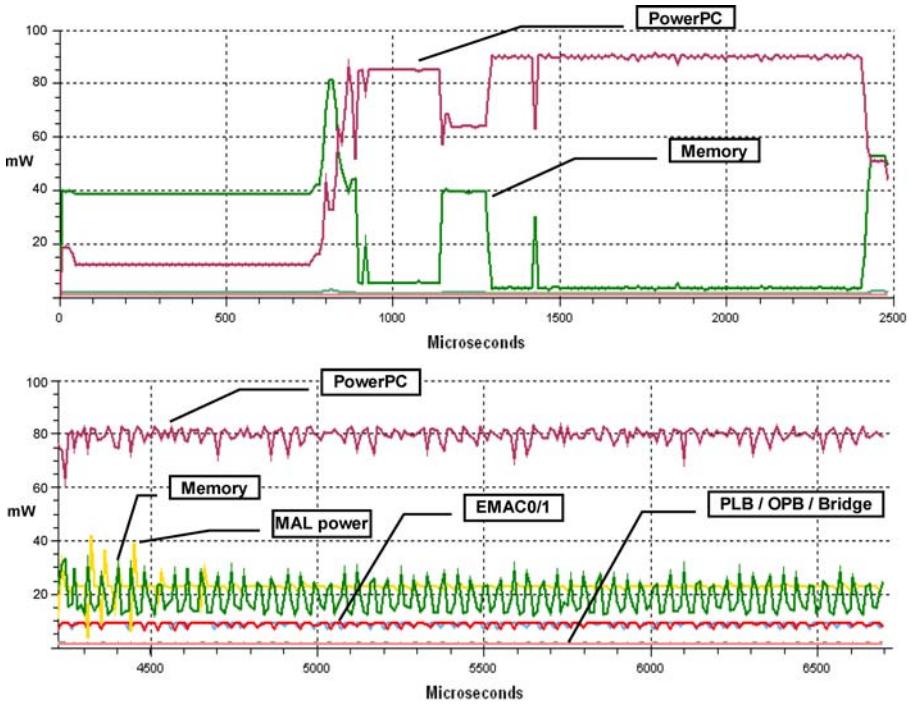


**Fig. 8** Average Power for different cores during execution of the Ethernet packet processing application

**Power estimation**

By turning on the power estimation capability during the execution of the application we are able to generate various types of power-related information and track them during the simulation, such as: (1) energy consumed per core and total, (2) energy consumed by each transaction and its duration, (3) average power per core, and (4) energy/power consumed by chosen functions (in the application code). Figure 8 plots the power consumed by the cores in this example using 512-byte packets, for a sampling interval of 10 microseconds, for two simulation intervals.

The top part of the figure plots the first 2500*us* of the simulation which is when the processor loads the code from memory and starts executing (but before any Ethernet packets are transferred). Until about 700*us* the operating system is initializing and the processor is loading instructions from memory and waiting for memory. The time between 700*us* and 1300*us* is spent mostly on filling up the packet buffer area in the cache and then writing to memory. The time between 1300*us* and 2400*us* is spent by the CPU setting up and initializing the packet descriptors, with no memory activity. The power on the other cores is negligible during these cycles. The bottom part of the figure shows the power during the beginning of the steady-state operation of the Ethernet subsystem. The processor continues to execute instructions and still represents the largest part of the power. Although the processor is largely underutilized, the application is such that it stays in a loop waiting for packets to handle, and thus consuming power. The second and third biggest contributors to the power are the MAL and memory. The MAL is active in transferring packets from memory to the EMACs over the PLB. The EMACs are also busy transferring/receiving data to/from the TX and RX

**Table 1** Power estimation comparison between TLMs and gate-level for different scenarios

| | Gate-level model | | Transaction-level model | | |
|---|---|---|---|---|---|
| | Power (mW) | Run time (min) | Power (mW) | Run time (min) | Error |
| Scenario-1 | 57.89 | 22.3 | 56.145 | 0.005 | −3.01% |
| Scenario-2 | 58.74 | 25.4 | 56.1194 | 0.01 | −4.46% |
| Scenario-3 | 58.595 | 26.8 | 57.071 | 0.02 | −2.6% |
| Scenario-4 | 22.744 | 35 | 21.975 | 0.02 | −3.38% |
| Scenario-5 | 57 | 45 | 63.35 | 1 | 11.19% |

FIFOs. Since this is mostly steady-state behavior, the power pattern on each of these cores is very repetitive. The lowest line in the bottom plot is the power for the PLB, OPB and the Bridge (the lines overlap), and it can be seen that they are very low when compared to the others. This type of power estimation can be very useful when analyzing the power patterns of a given application for power optimization.

### 6.2. Power analysis results

In order to validate our power modeling approach we devised a number of test cases involving specific types of data transfers. These test cases were designed to exercise different buses and peripherals, and analyze the effectiveness of our transaction level power estimation when multiple transactions take place in the system. We compared the accuracy and efficiency of the power estimation technique relative to the power computed using detailed gate-level simulation and power computation for the same SoC subsystem. The SoC sub-system used consisted of the following components: PLB bus and arbiter, OPB bus, External Bus Controller (EBC), PLB2OPB Bridge, DDR Memory Controller, Memory, and DMA Controller. Several scenarios representing different application behaviors were executed on the SoC sub-system. Since we were interested in validating the transaction-level power estimation, and not the processor power, the scenarios were described using a generic processor model that acts as a traffic generator for the rest of the cores on the subsystem. The total energy for each of the scenarios was collected from SystemC TLM simulation using the power models described in Section 4. The same scenarios that were executed by the generic processor model on the TLM platform were executed on the gate level version of the platform using the TOS setup, which converts these high level test cases representing the scenarios into vectors that drive the simulation of the gate level model of the subsystem. Table 1 shows the different scenarios, the computed power and runtime for both the TLM and gate-level simulations and power computation. All experiments were run on a 1GHz dual CPU Linux workstation.

*Scenario1: PLB TO PLB transfer.* This scenario represents a software initiated memory-to-memory transfer on the PLB bus. The three main steps are (1) CPU writes data to PLB address A; (2) DMA transfers data from PLB address A to B; and (3) CPU reads data from PLB address B and compares data with original data. 32 transfers of 128 bits each are executed.

*Scenario 2: PLB TO OPB transfer.* This is a software initiated memory-to-memory transfer between the PLB and OPB memories. OPB memory is connected to OPB BUS through an External Bus Controller (EBC). The PLB2OPB Bridge, OPB and EBC are all required to be active in order to read or write OPB memory. This involves the following steps: (1)

CPU transfers original data to PLB address A; (2) DMA transfers data from PLB address A to OPB address B; and (3) CPU reads data from OPB address B and compares it with original data. 32 transfers of 32 bits each are executed.

*Scenario 3: EBC (OPB) TO PLB transfer.* This transfer is from the memory connected to the EBC (which is connected to the OPB bus) to the memory on the PLB bus. The following steps occur: (1) CPU writes original data to OPB address A; (2) DMA transfers data from OPB address A to PLB address B; and (3) CPU reads data from PLB address B and compares with original data. 32 transfers of 32 bits each are executed.

*Scenario 4: EBC (OPB) TO EBC (OPB).* This test performs a software initiated memory-to-memory transfer between memories connected to the EBC on the OPB bus. The main steps are: (1) CPU writes original data to OPB address A; (2) DMA transfers data from OPB address A to B; and (3) CPU reads data from OPB address B and compares with original data. 32 transfers of 32 bits each are executed.

Scenarios 1 through 4 are cases where there is a high degree of correlation between the transaction level power models used and the transactions generated by the scenarios. The scenarios are simple transaction sequences generated by the cores involved with limited inter-transaction interaction. Although the overall system average power number error margin is considerably low, the error on individual core average power estimates can be slightly higher. For example, in Scenario-1 the differences between the gate level and transaction level core energy estimates for PLB, DMA, and DDRMC are 13%, −20%, and 4% respectively. The reason for the DMA controller showing a higher difference is because certain modes of operation of the controller (pre-fetch buffer enabled) were not captured in the HTLP-tree. The individual core average power data when combined together with idle times gives acceptable accuracy for the estimation. Moreover, the HTLP-tree can be refined to reflect more accurately the transactions defined in the models.

*Scenario 5: Complex PLB to OPB transfer.* This is a software initiated memory-to-memory transfer between PLB memory and memory connected to the EBC on the OPB. It is similar to Scenario-2 but involving multiple DMA transfers. Initially the necessary memory banks are setup by programming the DDR memory controller. This is followed by programming the DMA controller through the DCR bus, to software initiated memory to memory transfer mode. After these initialization transactions, the steps in this scenario are as follows: (1) initial data is written into multiple PLB addresses; (2) DMA transfers data from PLB address A to OPB address B; and (3) CPU reads data from OPB address B and compares it with original data. Step 2 is repeated with different source and destination addresses and is accompanied by a CPU read to verify the correctness of the DMA transfer. The higher error (11.19%) for this scenario may be due to potential interactions between transactions. These examples demonstrate that average power can be estimated using TLM with a reasonable degree of accuracy, which is well suited for early design analysis and exploration.

## 7. Conclusions

This paper presented the transaction-level modeling approach developed for simulation, verification and power analysis of systems-on-chip designed using the CoreConnect architecture. The overall approach for modeling transactions and power is general enough for other architectures. The quality of the models with respect to timing accuracy and power estimation were validated using realistic examples. A version of these models has been released through DeveloperWorks [18] and is available for download.

## References


1. Burton, M. and A. Donlin. Transaction-Level Modeling: Above RTL Design and Methodology. http://www.systemc.org.
2. AMBA - AHB Cycle Level Interface Specification. ARM white paper, available from http://www.arm.com.
3. Keutzer, K., S. Malik, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-Based Design. In *IEEE Transaction on Computer-Aided Design*, 19(12), December 2000.
4. Gajski, D., J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology.* Kluwer Academic Publishers, 2000.
5. The Open SystemC Initiative. http://www.systemc.org.
6. Grotker, T., S. Liao, G. Martin, and S. Swan . *System Design with SystemC*. Kluwer Academic Publishers, 2002.
7. CoreConnect™ Bus Architecture documents, available from http://www.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture.html.
8. SoC design with CoreConnect: 128-bit PLB explained. tutorial available from http://www.ibm.com/developerworks/edu/pa-dw-pa-socdesign-i.html.
9. Instruction-Set Simulator User's Guide. available from IBM under license at http://www.ibm.com /chips/power/licensing/
10. RISCWatch Debugger. links and documentation available from:http://www-03.ibm.com/chips/power/tools/riscwatc.html.
11. Benini, L., D. Bertozzi, D. Bruni, N. Drago, F. Fummi, M. Poncino. Legacy SystemC Co-Simulation of Multi-Processor Systems-on-Chip. In *Proceedings* 2002 *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, IEEE, 494, 2002.
12. Givargis, T.D., F. Vahid, and J. Henkel. Instruction-Based System-Level Power Evaluation of System-on-a-chip Peripherals. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS)*, 2000.
13. Caldari, M., M. Conti, M. Coppola, P. Crippa, S. Orcioni, L. Pieralisi, and C. Turchetti. System-Level Power Analysis Methodology Applied to the AMBA AHB Bus. DATE 2003.
14. Qu, G., N. Kawabe, K, Usame, and M. Potkonjak. Function-Level Power Estimation Methodology for Microprocessors. In *Proceedings of the 37th DAC*, 2000.
15. Zhong, L., S. Ravi, A. Raghunathan, and N.K. Jha. Power Estimation for Cycle-Accurate Functional Descriptions of Hardware. In *Proceedings of ICCAD*, 2004.
16. Shafi, H., P.J. Bohrer, J. Phelan, C.A. Rusu, and J.L. Peterson. Design and Validation of a Performance and Power Simulator for PowerPC systems. *IBM Journal of Research and Development*, 47(5/6), September/November 2003.
17. Devins, R. SoC Verification Software—Test Operating System. I*EEE/DATC Electronic Design Processes Workshop*, April 2001.
18. Meet the PowerPC 405 Evaluation Kit. DeveloperWorks web site http://www.ibm.com/developerworks/library/pa-pek/index.html.
19. Dhanwada, N., I. Lin, and V. Narayanan. A Power Estimation Methodology for SystemC Transaction Level Models. In *Proceedings of CODES+ISSS*, 2005.