Fast Exploration of Bus-Based Communication Architectures at the CCATB Abstraction

SUDEEP PASRICHA and NIKIL DUTT
University of California, Irvine
and
MOHAMED BEN-ROMDHANE
Newport Media Inc.

Currently, system-on-chip (SoC) designs are becoming increasingly complex, with more and more components being integrated into a single SoC design. Communication between these components is increasingly dominating critical system paths and frequently becomes the source of performance bottlenecks. It, therefore, becomes imperative for designers to explore the communication space early in the design flow. Traditionally, system designers have used Pin-Accurate Bus Cycle Accurate (PA-BCA) models for early communication space exploration. These models capture all of the bus signals and strictly maintain cycle accuracy, which is useful for reliable performance exploration but results in slow simulation speeds for complex, designs, even when they are modeled using high-level languages. Recently, there have been several efforts to use the Transaction-Level Modeling (TLM) paradigm for improving simulation performance in BCA models. However, these transaction-based BCA (T-BCA) models capture a lot of details that can be eliminated when exploring communication architectures. In this paper, we extend the TLM approach and propose a new transaction-based modeling abstraction level (CCATB) to explore the communication design space. Our abstraction level bridges the gap between the TLM and BCA levels, and yields an average performance speedup of 120% over PA-BCA and 67% over T-BCA models, on average. The CCATB models are not only faster to simulate, but also extremely accurate and take less time to model compared to both T-BCA and PA-BCA models. We describe the mechanisms that produce the speedup in CCATB models and also analyze how the achieved simulation speedup scales with design complexity. To demonstrate the effectiveness of using CCATB for exploration, we present communication space exploration case studies from the broadband communication and multimedia application domains.

Preliminary results of this work were presented at 2004 IEEE/ACM DAC [Pasricha et al., 2004a] and 2004 IEEE/ACM CODES+ISSS [Pasricha et al., 2004b] conferences.
This research was partially supported by grants from Conexant Systems Inc., UC Micro (03-029), CPCC fellowship and NSF grants CCR 0203813 and CCR 0205712.
Authors' addresses: Sudeep Pasricha and Nikil Dutt, Center for Embedded Computer Systems (CECS), University of California, Irvine, California 92697-3425; Mohamed Ben-Romdhane, Newport Media Inc., Lake Forest, California 92630. This work was performed while Dr. Ben-Romdhane was employed by Conexant, Inc.

Categories and Subject Descriptors: I.6.5 [**Computing Methodologies**]: Simulation and Modeling-Model Development—*Modeling methodologies*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques, Measurement techniques*; J.6 [**Computer Applications**]: Computer-aided Engineering—*Computer-aided design (CAD)*

General Terms: Performance, Measurement, Design

Additional Key Words and Phrases: System-on-chip, transaction-level modeling, communication architecture, on-chip bus, performance exploration

ACM Reference Format:

Pasricha, S., Dutt, N., and Ben-Romdhane, M. 2008. Fast exploration of bus-based communication architectures at the CCATB abstraction. ACM Trans. Embedd. Comput. Syst. 7, 2, Article 22 (February 2008), 32 pages. DOI = 10.1145/1331331.1331346 http://doi.acm.org/10.1145/1331331.1331346

1. INTRODUCTION

Today system-on-chip (SoC) designers are dealing with ever-increasing design complexity. SoC designs today have several components (CPUs, memories, peripherals, DSPs, etc.) that share the processing load and frequently exchange data with each other. Intercomponent communication is often in the critical path of a SoC design and is a very common source of performance bottlenecks [Davis and Meindl 1998; Sylvester and Keutzer 1998]. It, therefore, becomes imperative for system designers to focus on exploring the communication space quickly, reliably, and early in the design flow to make the right choices and eliminate performance bottlenecks under time-to-market pressures.

Shared-bus based communication architectures such as ARM AMBA [Flynn 1997], Sonics MicroNetwork [Wingard 2001], IBM CoreConnect [Hofmann and Drerup, 2002] and STMicroelectronics STBus [Scandurra et al. 2003] are some of the popular choices for on-chip communication between components in current SoC designs. These bus architectures can be configured in several different ways, resulting in a vast exploration space that is prohibitive to explore at the RTL level. Not only is the RTL simulation speed too slow to allow adequate coverage of the large design space, but making small changes in the design can require considerable reengineering effort because of the highly complex nature of these systems. To overcome these limitations, system designers have raised the abstraction level of system models. Figure 1 shows the frequently used modeling abstraction levels for communication space exploration, usually captured with high-level languages, such as C/C++ [D&T Roundtable, 2001]. These high-level models give an early estimate of the system characteristics before committing to RTL development. In-Cycle Accurate (CA) models [Yim et al. 1997; Jang et al. 2004], system components (both masters and slaves) and the bus architecture are captured at a cycle and signal accurate level. While these models are extremely accurate, they are too time consuming to model and only provide a moderate speedup over RTL models. Pin-Accurate Bus Cycle Accurate (PA-BCA) models [Séméria and Ghosh 2000] capture the system at a higher abstraction level than CA models. Behavior inside components need not be scheduled at every cycle boundary, which enables rapid system prototyping and considerable simulation speedup over RTL. The component interface and the bus are still modeled at a cycle and pin accurate level, which enables

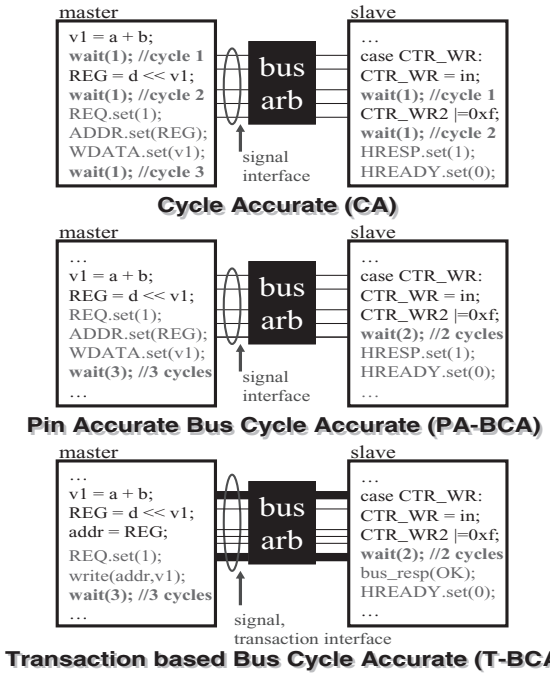


Fig. 1. Modeling abstractions for communication space exploration.

accurate communication space exploration. However, with the increasing role of embedded software and rising design complexity in modern SoC designs, even the simulation speedup gained with PA-BCA models is not enough. More recent research approaches [Zhu and Malik 2002; Caldari et al. 2003; Ogawa et al. 2003; AHB CLI, 2003] have focused on using concepts found in the Transaction-Level Modeling (TLM) [Gajski et al. 2000, Pasricha 2002; Grötker et al. 2002] domain (discussed in the next section) to speed up BCA model simulation even further with transaction-based BCA (T-BCA) models.

In this paper we introduce a new modeling abstraction level called (Cycle Count Accurate at Transaction Boundaries) (CCATB) for on-chip communication space exploration. Our abstraction level allows faster system prototyping and, more importantly, better simulation performance, while maintaining cycle count accuracy. CCATB models yield an average performance speedup of 120% over PA-BCA and 67% over T-BCA models. We describe the mechanisms behind the speedup and present a simulation implementation of the CCATB modeling abstraction, for high-performance shared bus architectures. To underline the effectiveness of our approach, we describe exploration case studies involving industrial strength SoC designs in the broadband communication and multimedia application domains. We also compare simulation performance and modeling effort for CCATB, PA-BCA, and T-BCA models and analyze the scalability of these approaches with design complexity.

The rest of the paper is organized as follows. Section 2 looks at some related work in this area. Section 3 briefly discusses requirements for a communication

design space exploration effort. Section 4 gives a brief overview of a commonly used bus architecture—AMBA. Section 5 describes in detail the CCATB modeling abstraction for exploring on-chip communication architectures. Section 6 presents an implementation of the CCATB simulation model and illustrates the sources of speedup. Section 7 describes two exploration case studies where we use CCATB models to explore the communication design space of SoC subsystems from the broadband communication and multimedia domains. Section 8 compares modeling effort and simulation speeds for the CCATB, T-BCA, and PA-BCA models, and shows how the speeds scale with increasing system complexity. Finally Section 9 concludes the paper and gives directions for future work.

2. RELATED WORK

Transaction-level models [Gajski et al. 2000, Pasricha 2002; Grötker et al. 2002] are bit-accurate models of a system with specifics of the bus protocol replaced by a generic bus (or *channel*) and where communication takes place when components call *read()* and *write()* methods provided by the channel interface. Since detailed timing and pin-accuracy are omitted, these models are fast to simulate and are useful for early functional validation of the system [Pasricha 2002]. Gajski et al. [2000] also proposed a top-down system design methodology with four models at different abstraction levels. The *architecture* model in their methodology corresponds to the TLM level of abstraction while the next lower abstraction level (called the *communication* model) is a BCA model where the generic channel has been replaced by bit and timing accurate signals corresponding to a specific bus protocol.

Early work with TLM established SystemC 2.0 [Grötker et al. 2002] as the modeling language of choice for the approach. In [Pasricha 2002] we described how TLM can be used for early system prototyping and embedded software development. Paulin et al. [2002] define a system-level exploration platform for network processors that need to handle high-speed packet processing. The SOCP channel described in their approach is based on OCP [OCP] semantics and is essentially a simple TLM channel with a few added details, such as support for split transactions [Flynn 1997]. Nicolescu et al. [2001] propose a component-based bottom-up system design methodology where components modeled at different abstractions are connected together with a generic channel, like the one used in TLM, after encapsulating them with suitable wrappers. Commercial tools such as the Incisive Verification Platform [NCSystemC], ConvergenSC System Designer [Coware], and Synopsys System Studio [System Studio] have also started adding support for system modeling at the higher TLM abstraction, in addition to lower-level RTL modeling. Finally, the OCP-IP specification [OCP] describes different layers of abstraction, to capture a communication system at varying levels of detail. Layer-0 is an extremely detailed pin, bit, and cycle accurate model; layer 1 is a transaction-based bus cycle accurate (T-BCA) abstraction; layer 2 is an approximately timed transaction-level model (T-TLM), which uses approximate timing and does not capture bus protocol details; layer 3 is an untimed transaction-level model (UT-TLM).

Recently, research efforts [Zhu and Malik 2002; Caldari et al. 2003; Ogawa et al. 2003; AHB CLI, 2003] have focused on adapting TLM concepts to speed up architecture exploration. Zhu and Malik [2002] use function calls instead of slower signal semantics to describe models of AMBA 2.0 and CoreConnect bus architectures at a high abstraction level. However, the resulting models are not detailed enough for accurate communication exploration. Caldari et al. [2003] similarly attempt to model AMBA 2.0 using function calls for reads/writes on the bus, but also model certain bus signals and make extensive use of SystemC *clocked threads,* which can slow down simulation. Ogawa et al. [2003] also model data transfers in AMBA 2.0 using read/write transactions, but use low-level handshaking semantics in the models that need not be explicitly modeled to preserve cycle accuracy. Recently, ARM released the AHB Cycle-Level Interface Specification [AHB CLI] which provides the definition and compliance requirements for modeling AHB at a cycle-accurate level in SystemC. Function calls are used to replace all bus signals at the interface between components and the bus. Although using function calls speeds up simulation, there is a lot of opportunity for improvement by reducing the number of calls while maintaining cycle accuracy, as we show later in this paper.

3. REQUIREMENTS FOR COMMUNICATION SPACE EXPLORATION

After system designers have performed hardware/software partitioning and architecture mapping in a typical design flow [Gajski et al. 2000], they need to select a communication architecture for the design. The selection is complicated by the plethora of choices [Flynn 1997; Wingard 2001, Hofmann and Drerup, 2002; Scandurra et al. 2003] that a designer is confronted with. Factors such as application domain-specific communication requirements and reuse of the existing design IP library play a major role in this selection process. Once a choice of communication architecture is made, the next challenge is to configure the architecture to meet design performance requirements. Bus-based communication architectures, such as AMBA, have several parameters that can be configured to improve performance: bus topology, data bus width, arbitration protocols, DMA burst lengths, and buffer sizes have significant impact on system performance and must be considered by designers during exploration. In the exploration studies presented in this paper, we use our approach to select an appropriate communication architecture and also configure it once the selection process is completed.

Any meaningful exploration effort must be able to comprehensively capture the communication architecture and be able to simulate the effects of changing configurable parameters at a system level [Loghi et al. 2004]. This implies that we need to model the entire system and not just a portion of it. Fast simulation speed is also very essential when exploring large designs and the vast design space, in a timely manner. System components such as CPUs, memories, and peripherals need to be appropriately parameterized [Ben-Romdhane et al. 1996], annotated with timing details and modeled at a granularity that would capture their precise functionality, yet not weigh down simulation speed because of unnecessary detail. Existing components that have been written at different abstraction levels (e.g., pin-accurate interface-processor ISS models) should be

easily adapted to fit into the framework by writing an appropriate wrapper to interface with our bus model. Performance numbers would then be obtained by simulating the working of the entire system—including running embedded software on the CPU architecture model. An important long-term requirement would be the ease of reuse of these components, to amortize design effort over a range of architecture derivatives. Our bus model would be required to support all the advanced high-performance bus features, such as pipelined operation, hierarchy, SPLIT/RETRY transactions, out-of-order transaction completion, burst modes, exclusive (semaphore) access, and protection modes, etc. The bus interface to SoC components should be independent of the underlying architecture to allow effortless plug-and-play of different on-chip communication architectures (AMBA, STBus, CoreConnect, etc.). It should also be generic enough to ease refinement from higher-level (timing-independent) TLM models to lower-level cycle/pin-accurate models, and avoid modeling protocol signals because of simulation overhead—instead function calls should be used.

Ultimately, the exploration models need to be fast, accurate, and flexible, providing good simulation speed, overall cycle accuracy for reliable performance estimation, and the flexibility to seamlessly plug-and-run different bus architectures and reuse components, such as processors, memories, and peripherals.

4. AMBA: A TYPICAL BUS ARCHITECTURE

Bus architectures usually have a separate bus for high performance, high bandwidth components, and low-bandwidth high-latency peripherals. A typical example of one such bus architecture is AMBA [Flynn 1997], which is one of the most widely used on-chip bus architecture standards used to interconnect components in SoC designs. Since we use the AMBA architecture to describe and demonstrate the features and exploration capabilities of CCATB in subsequent sections, we give a brief overview of the standard here.

The AMBA 2.0 bus architecture consists of the AHB (advanced high-performance bus), APB (advanced peripheral bus), and ASB (advanced system bus) buses. The AHB bus is used for high-bandwidth and low, latency communication, primarily between CPU cores, high-performance peripherals, DMA controllers, on-chip memories, and interfaces, such as bridges to the slower APB bus. The APB is used to connect slower peripherals, such as timers, and UARTs, and uses a bridge to interface with the AHB. It is a simple bus that does not support the advanced features of the AHB bus. The ASB bus is an earlier version of the high-performance bus that has been superseded by AHB in current designs.

More recently, ARM announced the release of AMBA 3.0 [AMBA AXI] with the next generation of high-performance bus protocol called the advanced eXtensible interface (AXI). In the following subsections, we give a brief overview of the main features of the high-performance bus protocols in AMBA.

4.1 AMBA 2.0 AHB

The advanced high-performance bus (AHB) is a high-speed, high-bandwidth bus that supports multiple masters. AHB supports a multilayer bus

architecture to optimize system bandwidth and improve performance. It supports pipelined operations for high-speed memory and peripheral access without wasting precious bus cycles. Burst transfers allow optimal usage of memory interfaces by giving advance information of the nature of the transfers. AHB also allows split transactions, which maximize the use of the system bus bandwidth by enabling high-latency slaves to release the system bus during the dead time while the slave is completing its transaction. In addition, wide bus configurations from 32 up to 1024 bits wide are supported.

4.2 AMBA 3.0 AXI

The advanced eXtensible interface (AXI) has all the advanced features of the AHB bus, such as pipelined and burst transfers, multimaster configuration, and a wide data bus. In addition, it has support for separate read and write channels, unaligned data transfer using byte strobes, and improved burst mode operation (only the start address of the burst is broadcast on the address bus). An important feature of AXI is support for multiple outstanding transactions and out-of-order (OO) transaction completion. OO transaction completion allows slaves to relinquish control of the bus, complete received transactions in any order, and then request rearbitration so a response can be sent back to the master for the completed transaction. This can dramatically boost performance in high-performance systems by allowing better utilization of shared buses. AXI also provides enhanced protection support (secure/non-secure transactions), enhanced system cache/buffer support (pins for specifying write-back/write through attributes and allocation strategies), a FIXED-burst mode (for repeated access to the same location), and exclusive access support for semaphore-type operations.

5. CCATB OVERVIEW

As the previous section indicated, bus architectures, such as AMBA, have several parameters that can be configured to improve performance. Our goal is to improve simulation performance for reliable exploration of on-chip communication architectures as early as possible in the design flow.

5.1 Modeling Abstraction

To enable fast exploration of the communication design space, we introduce a novel modeling abstraction level that is "cycle accurate" when viewed at "transaction boundaries." For this reason, we call our model Cycle Count Accurate at Transaction Boundaries (CCATB). A transaction, in this context, refers to a *read* or *write* operation issued by a master to a slave, that can either be a single data word or a multiple data burst transfer. Transactions at the CCATB level are similar to transactions at the TLM level [Pasricha 2002] except that we, in addition, pass bus protocol-specific control and timing information. Unlike PA-BCA and T-BCA models, we do not maintain accuracy at every cycle boundary. Instead, we raise the modeling abstraction and maintain cycle count accuracy at transaction boundaries, i.e., the number of bus cycles that elapse at the end of a transaction is the same when compared to cycles elapsed in a

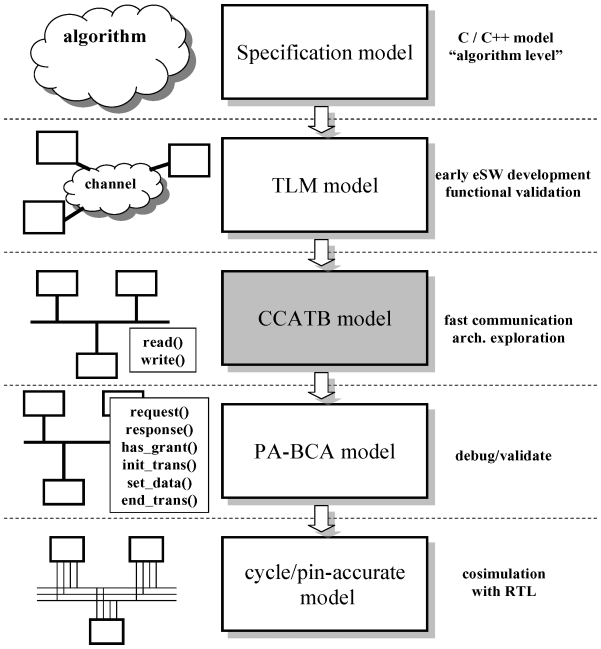


Fig. 2. System design flow with CCATB.

detailed cycle/pin accurate system model. A similar concept can be found in [Bergamaschi and Raje 1996], where observable time windows were defined and used for verifying results of high-level synthesis. We maintain overall cycle count accuracy needed to gather statistics for accurate communication space exploration, while optimizing the models for faster simulation. Our approach essentially trades off intratransaction visibility to gain simulation speedup.

5.2 Modeling Language

We chose SystemC 2.0 [Pasricha 2002, Grötker et al. 2002] to capture designs at the CCATB abstraction level, as it provides a rich set of primitives for modeling communication and synchronization—channels, ports, interfaces, events, signals, and wait-state insertion. Concurrent execution is performed by multiple threads and processes (light-weight threads) and execution schedule is governed by the scheduler. SystemC also supports capture of a wide range of modeling abstractions from high-level specifications to pin and timing accurate system models. Since it is a library based on C++, it is object oriented, modular, and allows data encapsulation—all of which are essential for easing IP distribution, reuse, and adaptability across different modeling abstraction levels.

5.3 Integrating CCATB in a System Design Flow

We define a modeling methodology which integrates our CCATB model in a high-level system design flow. Figure 2 depicts our proposed flow which has five system models at different abstraction levels.

At the topmost level is a specification model, which is a high-level algorithmic implementation of the functionality of the system. This model is generally captured in C or C++ and is independent of the hardware architecture that would eventually be used to implement the algorithm. After selecting available hardware components and partitioning functionality between hardware and software, we arrive at the *TLM* model ported to SystemC. At this level, high-level functional blocks representing hardware components, such as CPUs, memories, and peripherals are connected together using a bus architecture-independent generic channel. This system model is used for early embedded software development and high-level platform validation. It is generally untimed, but the model can be annotated with some timing information if a high-level estimate of system performance is required. Once the bus architecture is decided, the channels are merged onto a bus topology, each bus is annotated with timing and protocol details, and the interface is refined to obtain the *CCATB* model. This model is used for fast communication space and system performance exploration. The *read()*, *write()* channel interface from the *TLM* level remains the same as explained earlier—except that now bus architecture-specific control information also needs to be passed. Components from the *TLM* level can be easily and quickly refined to add this detail. If observable cycle accuracy for early system debugging and validation is required, the *read()* and *write()* interface calls can be decomposed into function calls, which consist of bus pins in the *PA-BCA* model. This is a bus cycle-accurate model, where the state of the system on the bus is accurately observable at every cycle boundary, instead of only at transaction boundaries like in the *CCATB* model. Finally, the components are refined further to obtain *pin/cycle-accurate* models, which can be manually or automatically mapped to RTL, or simply be used to cosimulate with existing RTL components for better simulation performance while validating system design at a low level.

5.4 Component Model Characteristics

Bus architectures in CCATB are modeled by extending the generic TLM *channel* [Pasricha 2002] to include bus architecture-specific timing and protocol details. Arbiter and decoder modules are integrated with this channel model. Computation blocks (masters and slaves) are modeled at the behavioral abstraction level, just like TLM models in [Pasricha 2002].

Masters are active blocks with possibly several computation threads and ports to interface with buses. One of our goals was to keep a consistent interface when refining models from the TLM level down to our CCATB level (Figure 2). Figure 3 shows the interface used by the master to communicate with a slave. In the figure, *port* specifies the port to send the read/write request on (since a master may be connected to multiple buses). *addr* is the address of the slave to send the transaction to. *token* is a structure that contains pointers to data and control information. Table I shows the main fields in this token data structure passed by the master and received by the arbiter. The *status* field in the token structure contains the status of the transaction, as returned by the slave. At the TLM level, since the bus is modeled as an abstract channel

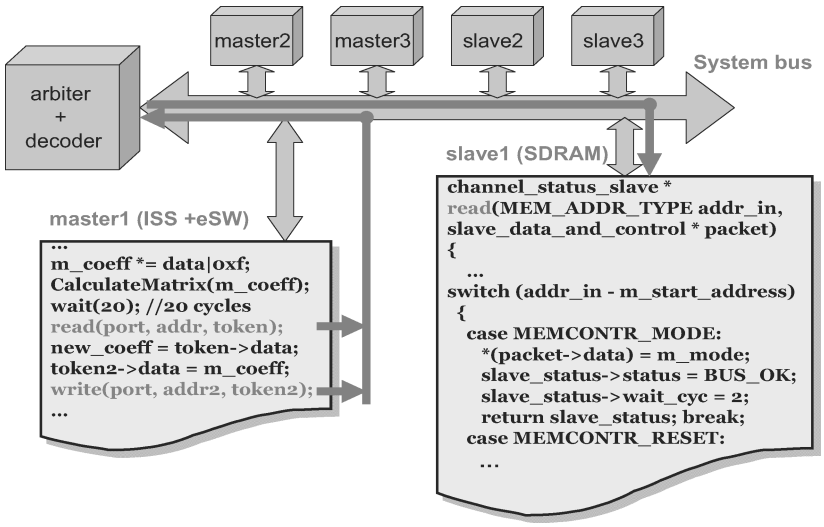


Fig. 3. CCATB transaction example.

Table I. Fields in *token* Structure

Request Field	Description
m_data	Pointer to an array of data
m_burst_length	Length of transaction burst
m_burst_type	Type of burst (incr, fixed, wrapping etc.)
m_byte_enable	Byte enable strobe for unaligned transfers
m_read	Indicates whether transaction is read/write
m_lock	Lock bus during transaction
m_cache	Cache/buffer hints
m_prot	Protection modes
m_transID	Transaction ID (needed for OO access)
m_busy_idle	Schedule of busy/idle cycles from master
m_ID	ID for identifying the master
status	Status of transaction (returned by slave)

without including any specific details of the bus protocol, the *data_cntrl* structure contains just the *m_data*, *m_burst_length*, and *m_byte_enable* fields. The other fields are specific to bus protocols and are thus omitted, since we are only concerned with transferring data packets from the source to its destination at this level. Thus, when we refine a master IP from the TLM level to the CCATB level, the only change is to set protocol-specific parameters before calling the interface functions.

Slaves are passive entities, activated only when triggered by the arbiter on a request from the master and have a register/memory map to handle read/write requests. The arbiter calls *read()* and *write()* functions implemented in the slave, as shown for the SDRAM controller in the figure. An excerpt of the read function from a memory controller is shown in Figure 3. Slaves can also have optional (light-weight) processes triggered by SystemC *events*, to perform computation if needed. The functionality of the slave IP remains unchanged when refining

the model from the TLM level to the CCATB level, unless the slave IP supports special bus protocol-specific features, such as having an outstanding instruction queue for out-of-order transaction completion in the AXI protocol, in which case, these details need to be added.

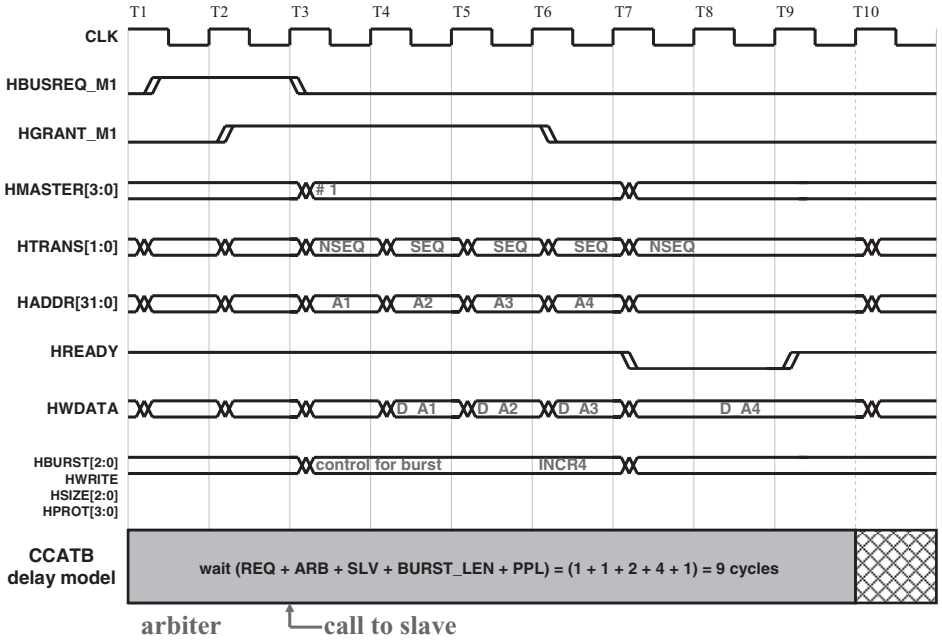
In accordance with the principle of interface-based design [Rowson and Sangiovanni-Vincentelli 1997], preexisting master and slave IP modules with different interfaces can be incorporated in the model using an adapter written in SystemC. For instance, we used adapter code written in SystemC in our exploration environment to interface ARM processor ISS models (which are not written in SystemC) with the TLM/CCATB SystemC interface.

5.5 Maintaining Cycle Count Accuracy

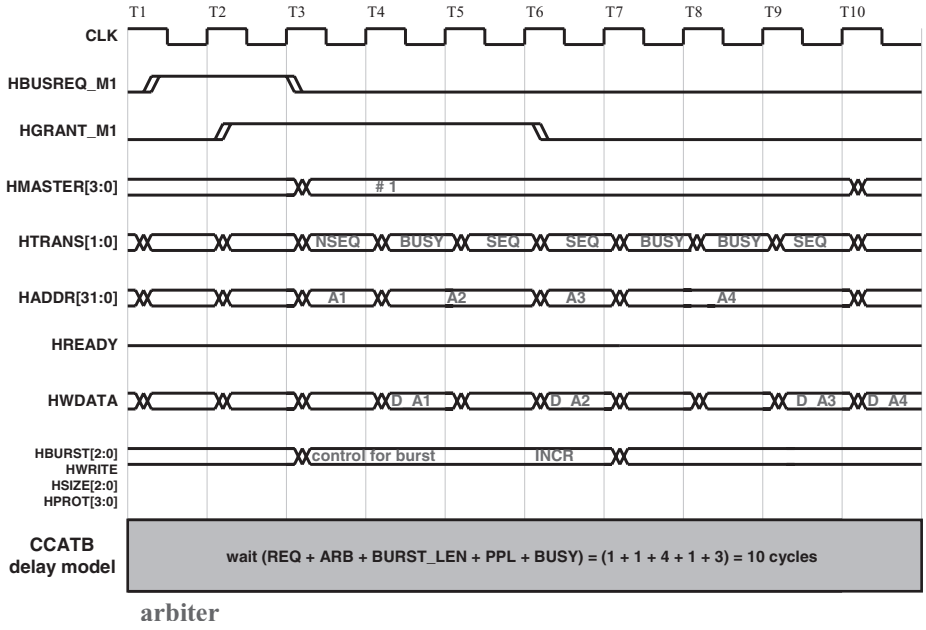
We will now illustrate how the CCATB model maintains cycle-count accuracy at transaction boundaries for different call sequences of the AMBA 2.0 protocol. First, we will describe the AMBA 2.0 signals used in these examples and then go into the details of the examples shown in Figures 4a, b, and c.

In AMBA 2.0, when a master needs to send or receive data, it requests the arbiter for access to the bus by raising the *HBUSREQx* signal. The arbiter, in turn, responds to the master via the *HGRANTx* signal. Depending on which master gains access to the bus, the arbiter drives the *HMASTERx* signals to indicate which master has access to the bus (this information is used by certain slaves). When a slave is ready to be accessed by a master, it drives the *HREADYx* signal high. Only when a master has received a bus grant from the arbiter via *HGRANTx* and detects a high *HREADY* signal from the destination slave, will it initiate the transaction. The transaction consists of the master driving the *HTRANSx* signal, which describes the type of transaction (sequential or nonsequential), the *HADDRx* signals, which are used to specify the slave addresses, and *HWDATAx*, if there is write data to be sent to the slave. Any data to be read from the slave appears on the *HRDATAx* signal lines. The master also drives control information about the data transaction on other signal lines—*HSIZEx* (size of the data item being sent), *HBURSTx* (number of data items being transferred in a burst transaction), *HWRITE* (whether the transfer is a read or a write), and *HPROTx* (contains protection information for slaves, which might require it).

We now describe the examples shown in Figure 4. In the first example in Figure 4a, a master requests an incremental write burst of length-4 data packets and the arbiter immediately grants it access to the bus. The transaction is initiated and data sent to the slave, but before it can process the final data packet in the sequence, the slave needs to perform involved computation with the previously written data that takes up two cycles. For this duration, the slave drives the *HREADY* signal low to indicate to the master that it is not ready yet to receive the final data packet in the burst. The burst transaction resumes once the slave drives *HREADY* high. The sequence of actions in the CCATB model is shown in Figure 5. The arbiter accounts for the request (*REQ*) and arbitration (*ARB*) delays for the write request before invoking the slave to complete the transaction. The slave performs the write and returns a token structure,

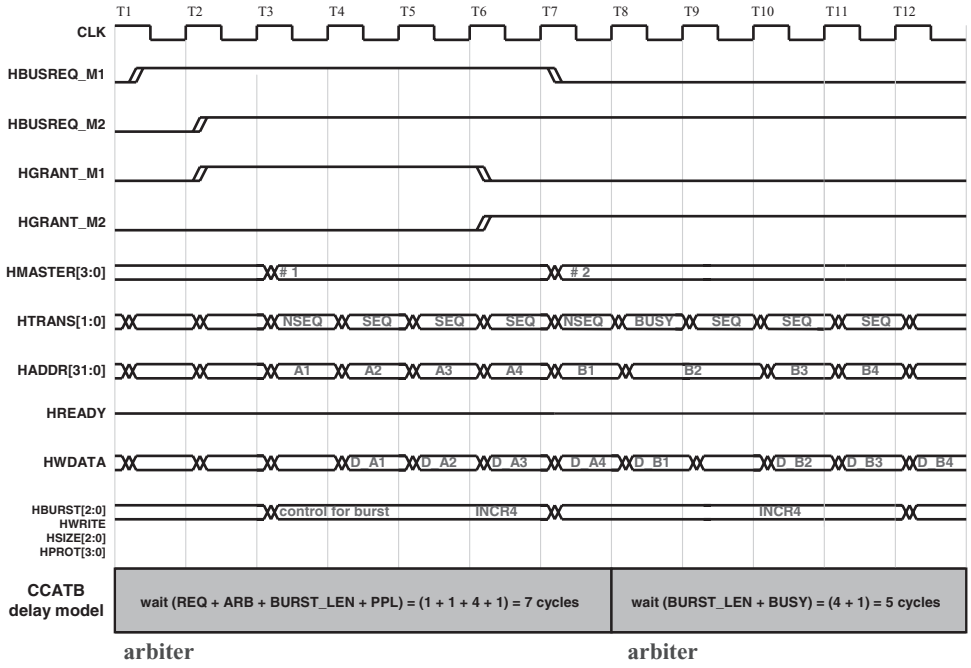


(a) AMBA 2.0 call sequence 1



(b) AMBA 2.0 call sequence 2

Fig. 4. *(Continues)*.



(c) AMBA 2.0 call sequence 3

Fig. 4. Reference AMBA 2.0 call sequences.

which contains the status of the write and an indication to the arbiter that two wait states need to be inserted. The arbiter then increments simulation time with the slave delay (*SLV*), burst length (*BURST_LEN*), and pipeline startup (*PPL*) delays. The arbiter then returns the status of the writes at the end of the transaction to the master.

Figure 4b illustrates a similar scenario, but, in this case, there is a delay in generating the data at the master end instead of a processing delay at the slave end. After the write burst initiates, the master indicates that it requires extra cycles to generate write data for the slave by sending a *BUSY* status on the *HTRANS[1:0]* lines. In the CCATB model, the arbiter gets a schedule of busy cycles from the master when it receives the transaction request, and thus it accounts for the *BUSY* cycle delay in the transaction, along with the other delays discussed above. There is no delay at the slave and consequently no increment in simulation time resulting from slave delay, in this case.

In Figure 4c, after a master requests access to the bus for a write burst, another master requests the bus for a write burst. While there is no delay at the master or the slave end for the first write burst, there is delay in generating the data at the master end for master M2, which is indicated by the *BUSY* status on the *HTRANS[1:0]* lines. In the CCATB model, the arbiter accounts for the *REQ*, *ARB*, *BURST_LEN*, and *PPL* delays and increments simulation time. For the subsequent transaction by master M2, the request has already been registered at the arbiter and no arbitration is required, so there is no *REQ* or *ARB* delay.

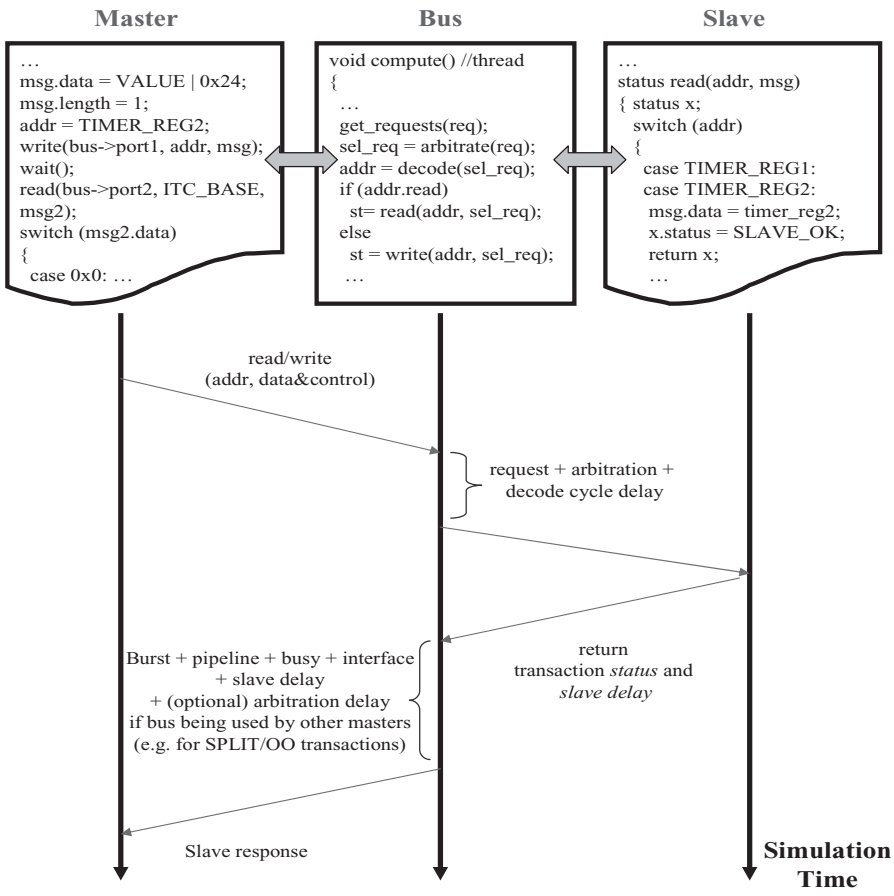


Fig. 5. CCATB transaction execution sequence.

Since transfers are pipelined, there is also no pipeline startup delay like in the case of master M1. Thus, there is no *PPL* delay. There is, however, delay, which is dependent on the burst length (*BURST_LEN*) and the busy cycles (*BUSY*), which is accounted for by the arbiter. Like in the previous scenario, the slave does not delay either of the burst transactions, so there is no simulation time increment because of slave delay.

6. SIMULATION SPEEDUP USING CCATB

We now describe an implementation of the CCATB simulation model to explain how we obtain simulation speedup. We consider a design with several bus subsystems each with its own separate arbiter and decoder, and connected to the other subsystems via bridges. The bus subsystem supports pipelining, burst mode transfers, and out-of-order (OO) transaction completion, which are all features found in high-performance bus architectures, such as AMBA 3.0 AXI.

We begin with a few definitions. Each bus subsystem is characterized by a tuple set X, where $X = \{R_{pend}, R_{act}, R_{oo}\}$. R_{pend} is a set of read/write

requests pending in a bus subsystem, waiting for selection by the arbiter. R_{act} is a set of *read*/*write* requests actively executing in a subsystem. R_{oo} is a set of out-of-order *read*/*write* requests in a subsystem that are waiting to enter into the pending request set (R_{pend}) after the expiration of their *OO latency period* (number of cycles that elapse after the slave releases control of the bus and before it requests rearbitration). As mentioned previously in Section 4.2, OO transaction completion allows slaves to relinquish control of the bus, complete received transactions in any order, and then request rearbitration so a response can be sent back to the master for the completed transaction. We define A to be a superset of the sets X for all p bus subsystems in the entire system,

$$A = \bigcup_{i=1}^{p} Xi$$

Next we define τ to be a transaction request structure, issued by a master to a slave. In addition to the subfields in Table I, it also includes the following subfields:

- *wait_cyc*—specifies the number of wait cycles before the bus can signal transaction completion to the master
- *oo_cyc*—specifies the number of wait cycles before the request can apply for re-arbitration at the bus arbiter
- *ooflag*—indicates if the request is an out-of-order transaction

Let *status* be defined as a transaction response structure returned by the slave. It contains a field (*stat*) that indicates the status of the transaction (OK, ERROR, etc.) as well as fields for the various delays encountered, such as those for the slave interface (*slave_int_delay*), slave computation (*slave_comp_delay*), and bridges (*bridge_delay*).

Finally, let $M = \{m1, m2, \ldots, m_n\}$ be a set of all masters in the system. Each master is represented by a value in this set, which corresponds to the sum of (1) the number of cycles before the next read/write request is issued by the master and (2) the master interface delay cycles. These values are maintained in a global table with an entry for each master and do not need to be specified manually by a designer—a preprocessing stage can automatically insert directives in the code to update the table at the point when a master issues a request to a bus.

Our approach speeds up simulation by preventing unnecessary invocation of simulation components and efficiently handling idle time during simulation. We now describe the implementation for our simulation model to show how this is accomplished. Figure 6 gives a high-level overview of the flow for the CCATB simulation model. There are two main phases in the model—the first is triggered on a positive edge of the system clock, while the second is triggered on the negative edge of the system clock. In the first phase, on the positive edge of the clock, we gather all the read and write requests in the system (*GatherRequests*). In the second phase, on the negative edge of the clock, we process these requests by calling the *HandleBusRequests* procedure, which, in turn, calls various functions and subprocedures to perform different tasks. In the first step

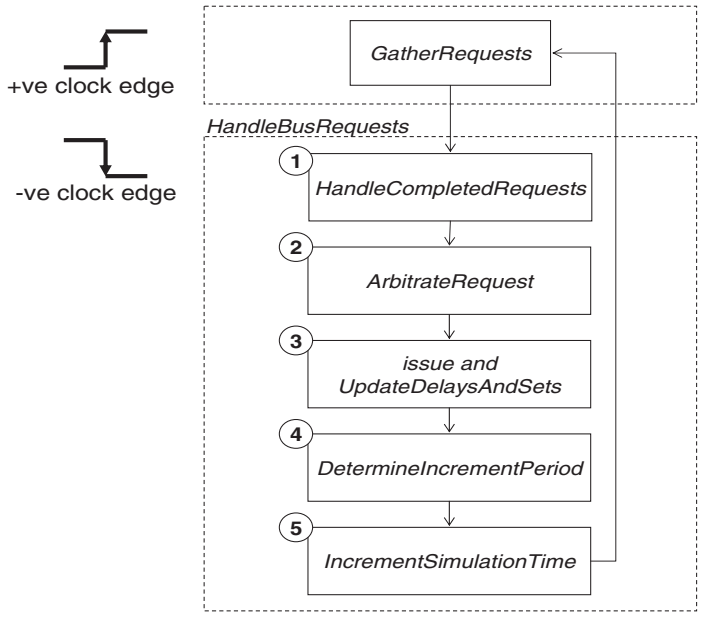


Fig. 6. CCATB simulation flow.

```
procedure GatherRequests()
begin
    if request then
        τ ⇐ request
        τ.wait_cyc ⇐ 0
        τ.oo_cyc ⇐ 0
        τ.ooflag ⇐ FALSE
        R_pend ⇐ R_pend ⋃ τ
end
```

Fig. 7. *GatherRequests* procedure.

inside the *HandleBusRequests* procedure, we handle completed requests and notify masters about completed transactions (*HandleCompletedRequests*). In the second step, we arbitrate on the buses in the system to select the requests, which have the highest priority and grant them access to their respective buses (*ArbitrateRequest*). In the third step, we issue the selected *read* or *write* requests to the slaves, receive a response from the slaves (*issue*), and update the request tokens with appropriate delay cycles for which we must wait before notifying the master (*UpdateDelaysAndSets*). In the fourth step, we determine the number of cycles to increment the simulation time (*DetermineIncrementPeriod*) before finally incrementing the simulation time (*IncrementSimulationTime*) in the fifth step. This completes one iteration cycle of the simulation (which could represent one or more *simulation* cycles based on the amount by which simulation time is incremented), which is repeated until the simulation ends.

We now describe the implementation of our simulation model in detail. Figure 7 shows the *GatherRequests* procedure from a bus module, which is

```
procedure HandleBusRequests()
begin
  for each set X ∈ A do
      HandleCompletedRequests(R_pend, R_act, R_oo)
      T ⇐ ArbitrateRequest(X, R_pend)
      for each request τ ∈ T do
          if ( τ.ooflag == TRUE) then
              R_act ⇐ R_act ⋃ τ
          else
              status ⇐ issue( τ.port, τ.addr, τ )
              UpdateDelaysAndSets(status, τ, R_act, R_oo)

  ψ ⇐ DetermineIncrementPeriod(A)
  IncrementSimulationTime( ψ, A, M )
end
```

Fig. 8. *HandleBusRequests* procedure.

```
procedure HandleCompletedRequests(R_pend, R_act, R_oo)
begin
  S_pend ⇐ R_pend ;  S_act ⇐ null ;  S_oo ⇐ null ;
  for each request τ ∈ R_act do
      if ( τ.wait_cyc == 0) then
          notify( τ.master, τ.status)
      else
          S_act ⇐ S_act ⋃ τ
  for each request τ ∈ R_oo do
      if ( τ.oo_cyc == 0) then
          S_pend ⇐ S_pend ⋃ τ
      else
          S_oo ⇐ S_oo ⋃ τ
  R_pend ⇐ S_pend ;  R_act ⇐ S_act ;  R_oo ⇐ S_oo ;
end
```

Fig. 9. *HandleCompletedRequests* procedure.

triggered on a positive clock edge, for every *read* or *write* transaction request issued by a master connected to that bus. *GatherRequests* simply adds the transaction request to the set of pending requests R_{pend} for the bus subsystem.

On the negative clock edge, the *HandleBusRequests* procedure (Figure 8) in the bus module is triggered and calls several subprocedures and functions to handle the communication requests in the system. *HandleBusRequests* first calls the *HandleCompletedRequests* subprocedure (Figure 9) for a bus subsystem X, to check if any executing requests in R_{act} have completed, in which case the appropriate master is notified and the transaction completed. *HandleCompletedRequests* also removes an out-of-order request from the set of out-of-order requests R_{oo} and adds it to the pending request set R_{pend} if it has completed waiting for its specified *OO latency period*.

Next, we arbitrate to select requests from the pending request set R_{pend}, which will be granted access to a bus in a bus subsystem X. The function *ArbitrateRequest* (Figure 10) performs the selection based on the arbitration policy selected for every bus. We assume that a call to the *ArbitrateOnPolicy*

```
function ArbitrateRequest(X, R_pend)
begin
  T ⇐ null
  for each independent channel c ∈ X do
      T ⇐ T ⋃ ArbitrateOnPolicy(c, R_pend)
  R_pend ⇐ R_pend \ T
  return T
end
```

Fig. 10. *ArbitrateRequest* function.

function applies the appropriate arbitration policy and returns the selected requests for the bus. After the selection, we update the set of pending requests R_{pend} by removing the requests selected for execution (and, hence, not "pending" anymore). Since a bus subsystem can have independent read and write channels [Rowson and Sangiovanni-Vincentelli 1997], there can be more than one active request executing in the subsystem, which is why *ArbitrateRequest* returns a set of requests and not just a single request for every subsystem.

After the call to *ArbitrateRequest* (Figure 10) from *HandleBusRequests* (Figure 8), if the *ooflag* field of the selected request is TRUE, it implies that this request has already been issued to the slave and now needs to wait for $\tau.wait\_cyc$ cycles before returning a response to the master. Therefore, we simply add it to the executing requests set R_{act}. Otherwise, we *issue* the request to the slave, which completes the transaction in zero-time and returns a status to the bus module. We use the returned *status* structure to update the transaction status by calling the *UpdateDelaysAndSets* procedure.

Figure 11 shows the *UpdateDelaysAndSets* procedure. In this procedure we first check for the returned error status. If there is no error, then depending on whether the request is an out-of-order type or not, we update $\tau.oo\_cyc$ with the number of cycles to wait before applying for rearbitration, and $\tau.wait\_cyc$ with the number of cycles before returning a response to the master. We also update the set R_{act} with the actively executing requests and R_{oo} with the OO requests. If an error occurs, then the actual slave computation delay can differ and is given by the field *error_delay*. The values for other delays, such as burst length and busy cycle delays, are also adjusted to reflect the truncation of the request because of the error.

After returning from the *UpdateDelaysAndSets* procedure in *HandleBusRequests* (Figure 8), we find the minimum number of cycles (ψ) before we need to iterate through the simulation flow (Figure 6) again (i.e., invoke the *Gather Requests* and *HandleBusRequests* procedures again), by calling the *DetermineIncrementPeriod* function (Figure 12). This function returns the minimum value out of the wait cycles for every executing request ($\tau.wait\_cyc$), out-of-order request cycles for all waiting OO requests ($\tau.oo\_cyc$), and the next request latency cycles for every master (λ). If there is a pending request that needs to be serviced in the next cycle, the function returns 1, which is the worst-case return value.

Finally, the increment value ψ returned by *DetermineIncrementPeriod* in *HandleBusRequests* (Figure 8) is used to update simulation time by calling the *IncrementSimulationTime* procedure, shown in Figure 13. By default, the

```
procedure UpdateDelaysAndSets(status, τ, R_act, R_oo)
begin
  if (status.stat == OK) then
      τ.status = OK
      if (status.oo == TRUE) then
          τ.ooflag ⇐ TRUE
          τ.oo_cyc ⇐ status.(oo_delay + slave_int_delay
                     + slave_comp_delay + bridge_delay)
                     + τ.arb_delay
          τ.wait_cyc ⇐ τ.(busy_delay + burst_length_delay
                     + ppl_delay + bridge_delay + arb_delay)
          R_oo ⇐ R_oo ⋃ τ
      else
          τ.wait_cyc ⇐ status.(slave_int_delay
                       + slave_comp_delay + bridge_delay)
                       + τ.(busy_delay + burst_length_delay
                       + ppl_delay + arb_delay)
          R_act ⇐ R_act ⋃ τ
  else
      τ.status = ERROR
      τ.wait_cyc ⇐ status.(slave_int_delay
                   + bridge_delay + error_delay)
                   + τ.(busy_delay + burst_length_delay
                   + ppl_delay + arb_delay)

end
```

Fig. 11. *UpdateDelaysAndSets* procedure.

```
function DetermineIncrementPeriod(A)
begin
  ψ ⇐ inf
    for each set X ∈ A do
        for each set R_pend ∈ X do
            if R_pend ≠ NULL then
                ψ ⇐ 1
                return ψ
        for each set R_act ∈ X do
            for each request τ` ∈ R_act do
                ψ ⇐ min { ψ, τ`.wait_cyc }
        for each set R_oo ∈ X do
            for each request τ`` ∈ R_oo do
                ψ ⇐ min { ψ, τ``.oo_cyc }
    for each value λ ∈ M do
        ψ ⇐ min { ψ, λ }
  return ψ
end
```

Fig. 12. *DetermineIncrementPeriod* function.

```
procedure IncrementSimulationTime( ψ, A, M )
begin
    for each set X ∈ A do
        for each request τ ∈ R_oo do
            τ.oo_cyc ⇐ τ.oo_cyc - ψ
        for each request τ ∈ R_act do
            τ.wait_cyc ⇐ τ.wait_cyc - ψ
    for each value λ ∈ M do
        λ ⇐ λ - ψ
    simulation_time ⇐ simulation_time + ψ
end
```

Fig. 13. *IncrementSimulationTime* procedure.

simulation flow incorporating the *GatherRequests* and *HandleBusRequests* procedures (shown in Figure 6) is invoked for every simulation cycle, but if we find a value of ψ, which is greater than 1, we can safely increment system simulation time by that value, preventing unnecessary iterations of the simulation flow and unnecessary invocations of procedures, thus speeding up simulation.

It should be noted that for some very high-performance designs, it is possible that there is very little scope for this kind of speedup. Although this might appear to be a limitation, there is still substantial speedup achieved over T-BCA and PA-BCA models, because we handle all the delays in a transaction in one place—in the bus module—without repeatedly invoking other parts of the system on every cycle (master and slave threads and processes), which would otherwise contribute to simulation overhead.

In the next two sections, we will present some experiments with the CCATB modeling abstraction. Section 7 presents exploration case studies with a broadband communication and a multimedia SoC design. The experiments on these two case studies show how various aspects of the communication architecture design space affect system performance and why it is important to consider them during a design space exploration effort. This motivates the need for a modeling abstraction, such as CCATB, that can (1) quickly capture the communication architecture design, (2) accurately capture delays and parameters of the communication architecture, which impact system performance, and (3) allow fast simulation speeds for comprehensive design space exploration. Subsequently in Section 8, we change the focus to a comparison of the CCATB approach with other approaches used for communication architecture exploration, such as PA-BCA and T-BCA. The experiments in this section compare modeling time and simulation speedup, and show how CCATB is a suitable abstraction for fast communication architecture design space exploration.

7. EXPLORATION CASE STUDIES

To demonstrate the effectiveness of exploration with CCATB, we present two case studies where we used CCATB models to explore the communication design space of the system. In the first case study, we compare and configure bus architectures for a SoC subsystem used in the broadband communication domain. In the second, we assume that the choice of bus architecture has already

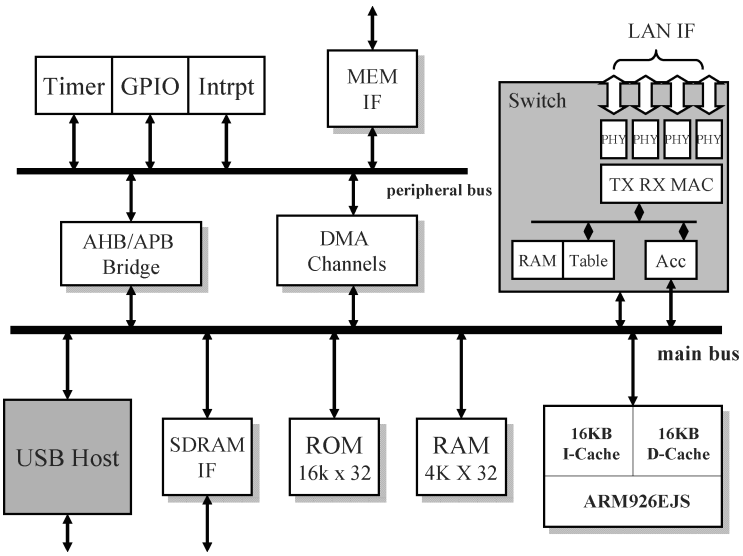


Fig. 14. Broadband communication SoC platform.

been made and we explore different configurations of the bus architecture for a multimedia SoC subsystem.

7.1 Case Study 1: Broadband Communication SoC

In this case study, we modeled an actual industrial strength SoC platform and performed several communication space exploration experiments on it. We present four of these in this section. All of these experiments were reproduced and verified at the more refined PA-BCA level. Figure 14 shows this SoC platform, which has applications in the broadband communication domain. We execute three proprietary benchmarks (*COMPLY, USBDRV*, and *SWITRN*) on the ARM926EJ-S processor instruction-set simulator (ISS), each of which activate different modes of operation for the platform. COMPLY configures the USB, switch, and DMA modules to drive traffic on the shared bus. USBDRV also configures the USB and DMA to drive traffic normally on the bus, but the switch activity is restricted. SWITRN configures the switch to drive traffic on the bus normally, but restricts USB and DMA activity.

In our first experiment, we attempted to observe the effect of changing communication protocol on overall system performance. We first simulated the platform with the AMBA 2.0 AHB system bus and then replaced it with the AMBA 3.0 AXI bus protocol, keeping the same driver application in both cases and without changing any bus parameters, such as arbitration strategy. Figure 15 shows that the AXI protocol improves overall system throughput compared to AHB. This is because in AMBA 2.0, the address bus is occupied mostly by transmission of addresses of transactions within a burst. In contrast, only the first address of a burst is transmitted in AMBA 3.0 AXI, which, coupled with transaction reordering, allows improved simultaneous read/write transaction execution and better throughput. Our model allows rapid plug-and-play

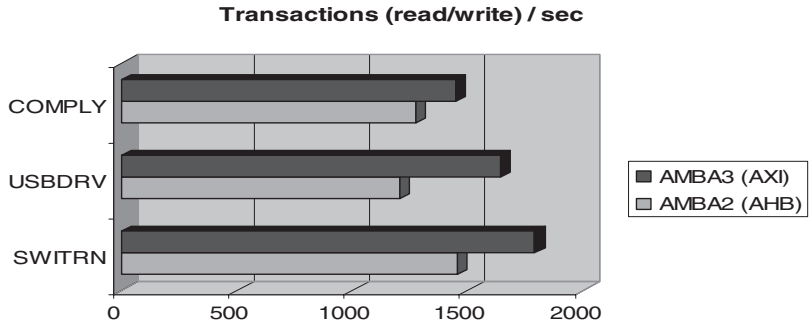


Fig. 15. Bus protocol comparison.

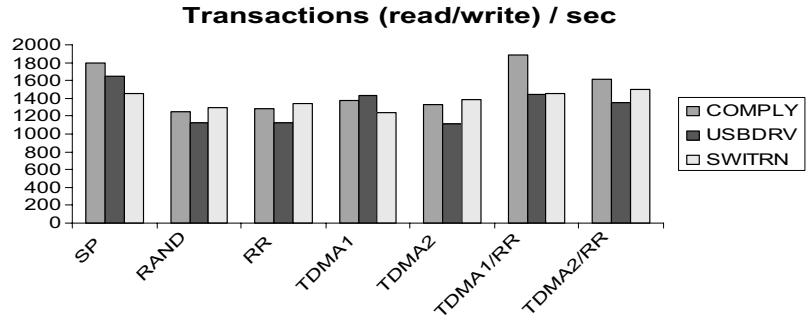


Fig. 16. Arbitration strategy comparison.

exploration of different bus architectures, requiring changes in just a few lines of code to declare and instantiate the bus in the top-level file.

Next, we explore the effect of arbitration strategies on system performance. We used the AMBA 2.0 AHB system bus and tested the following arbitration strategies—static priority (SP), random priority (RP), round robin (RR), time division multiple access or TDMA with two slots for the USB host and one for the rest (TDMA1), TDMA with two slots for the switch subsystem and one for the rest (TDMA2), TDMA1 with RR (TDMA1/RR) and TDMA2 with RR (TDMA2/RR), where the RR strategy is applied only if the selected master has no transaction to issue. Figure 16 shows the bus throughput for the three benchmarks. It can be seen that TDMA1/RR outperforms other schemes for COMPLY, while static priority works best for USBDRV (with the USB host given the maximum priority) and SWITRN (where the switch subsystem is given the maximum priority). We measure overall bus throughput—however, if bandwidth constraints for certain masters need to be met and overall throughput is a less important criteria, then other strategies might give better results. Also, more involved strategies such as a dynamic priority scheme can be easily introduced into this framework if traffic-based adaptable behavior is preferred.

To determine the influence of bus hierarchy on improving system performance by eliminating conflicts on a shared bus, we decomposed the shared bus into two hierarchical buses in our next experiment—in configuration A

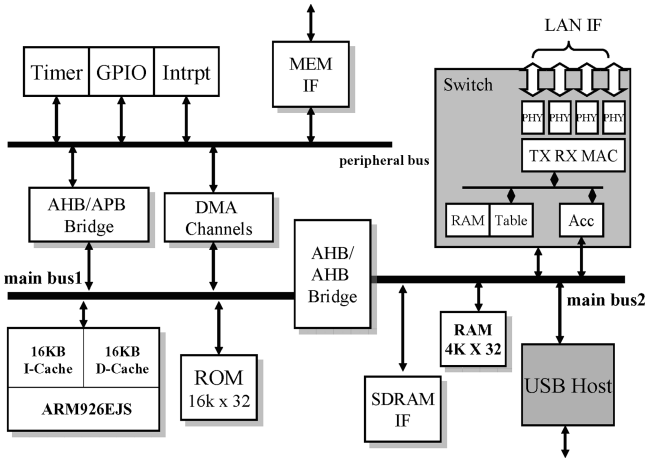


Fig. 17. Topology configuration A.

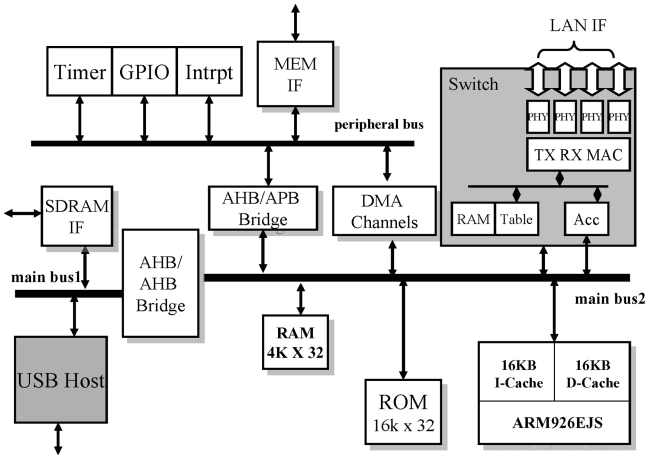


Fig. 18. Topology configuration B.

(Figure 17), we kept the ARM CPU and DMA master on one bus and the switch subsystem and USB host master on the other. In configuration B (Figure 18) we kept the ARM CPU, DMA and the switch subsystem on one bus while the USB host was given a dedicated bus. We used the TDMA1/RR strategy for conflict resolution. Figure 19 shows bus conflicts for these cases. It can be seen that configuration A has the least conflicts for COMPLY and SWITRN. This is because configuration A avoids conflicts between the DMA and the switch module, which is the main source of conflict in SWITRN and one of the main ones in COMPLY (along with the USB switch conflict). Configuration B is the best for USBDRV since conflicts between the USB (which drives the maximum traffic) and the DMA (which also drives a lot of traffic) are reduced when the USB is given a dedicated bus.

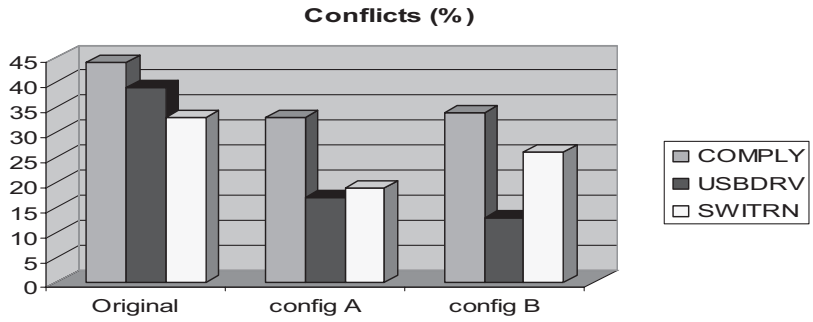


Fig. 19. Topology configuration comparison.

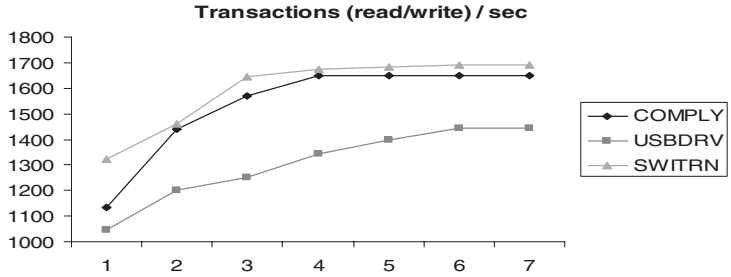


Fig. 20. Varying SDRAM OO queue size.

Finally, we study the effect of changing outstanding request queue size for the SDRAM IF module, which supports out-of-order execution of *read*/*write* requests as specified by the AMBA 3.0 AXI protocol. Figure 20 shows the effect of change in performance when the queue size is changed. It can be seen that performance saturates and no more gain can be obtained after the queue size has been increased to four for *COMPLY* and six for *SWITRN* and *USBDRV*. This is a limit on the number of simultaneous requests issued at any given time for the SDRAM IF by the masters in the system for these benchmarks. It can be seen that this parameter is highly application dependent and changes with changing application requirements, demonstrating the need for this type of an exploration environment.

7.2 Case Study 2: Multimedia SoC Subsystem

For our second case study, we explore a consumer multimedia SoC subsystem which performs audio and video encoding for popular codecs, such as MPEG. Figure 21 shows this platform, which is built around the AMBA 2.0 communication architecture. The system has an ARM926EJ-S processor with embedded software running on it to supervise flow control and perform encryption, a fast USB interface, on-chip memory modules, a DMA controller, an SDRAM controller to interface with external memory components, and standard peripherals, such as, a timer, UART, interrupt controller, general purpose I/O, and a compact flash card interface.

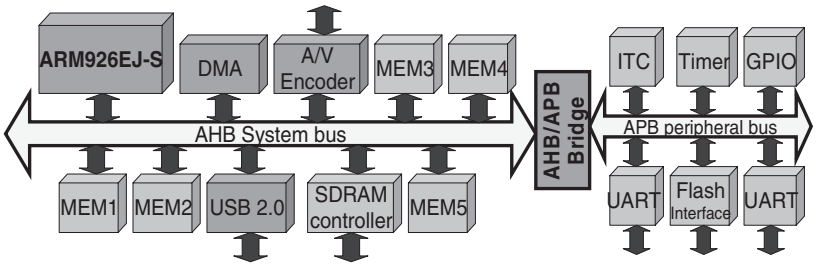
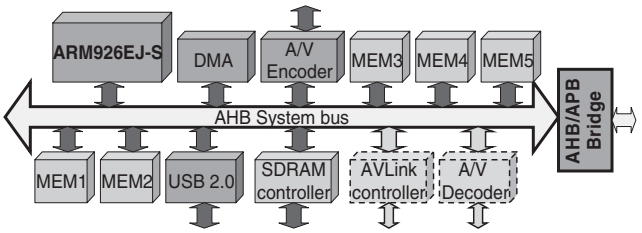


Fig. 21. SoC multimedia subsystem.

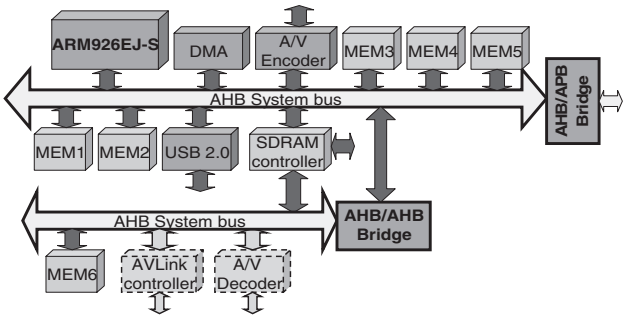
Consider a scenario where the designer wishes to extend the functionality of this encoder system to add support for audio/video decoding and an additional AVLink interface for streaming data. The final architecture must also meet peak bandwidth constraints for the USB component (480 Mbps) and the AVLink controller interface (768 Mbps). Figure 22a shows the system with the additional components added to the AHB bus. To explore the effects of changing communication architecture topology and arbitration protocols on system performance, we modeled the SoC platform at the CCATB level and simulated a test program for several interesting combinations of topology and arbitration strategies. For each configuration, we determined if bandwidth constraints were being met and iteratively modified the architecture until all the constraints were satisfied.

Table II shows the system performance (total cycle count for test program execution) for some of the architectures we considered-shown in Figure 22a–d). Column 2 in Table II shows the performance estimate for a high-level approximate-timed transaction-level model (T-TLM) of the system for comparison purposes, which uses behavioral components annotated with delays, just like the CCATB model, but uses bus protocol-independent *channels* with no contention for communication. This T-TLM model incidentally corresponds to the OCP layer-2 [OCP] approximate-timed transaction-level model, as described in Section 2. As a result of not accounting for the bus topology and protocol overhead, the T-TLM simulation completes execution of the test program in significantly fewer cycles, giving an incomplete and inaccurate estimate of system performance, with approximately a 400% error, compared to the more detailed CCATB model. The CCATB model is sensitive to changes in the bus topology and protocol, as can be seen with the varying execution performances for different topology and arbitration protocol configurations in Table II, and is thus a much more suitable abstraction for exploring the communication architecture space. In the next section, we will show how CCATB models are not only faster to simulate, but also to create, when compared to other abstractions currently being used for exploring the communication architecture space.

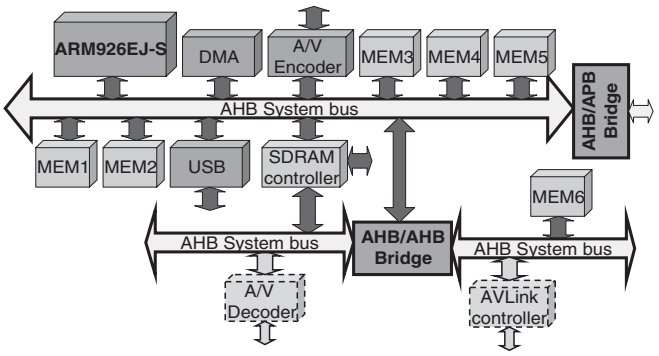
We will focus on the CCATB model for this exploration study from this point onward. In the columns for arbitration strategies, RR stands for a round-robin scheme where bus bandwidth is equally distributed among all the masters. TDMA1 refers to a TDMA strategy where in every frame four slots are allotted to the AVLink controller, two slots to the USB, and one slot for the remaining



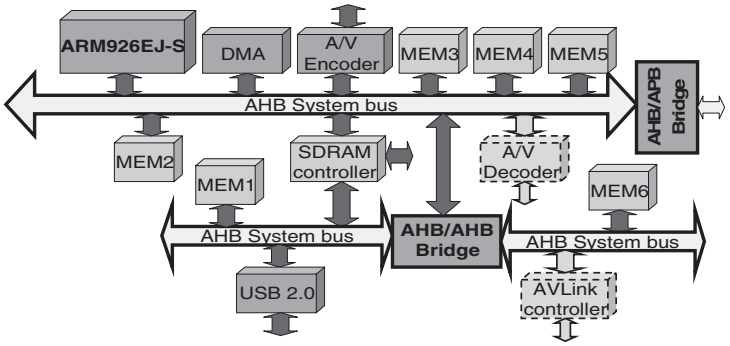
(a) Soc Architecture 1 (Arch1)



(b) Soc Architecture 2 (Arch2)



(c) Soc Architecture 3 (Arch3)



(d) Soc Architecture 4 (Arch4)

Fig. 22. SoC communication architecture topologies.

Table II. Execution Cycle Counts (in Millions of Cycles)

Architecture	T-TLM	CCATB with Different Arbitration Schemes				
		RR	TDMA1	TDMA2	SP1	SP2
Arch1	6.57	27.24	24.65	25.06	25.72	26.49
Arch2		24.98	23.86	23.03	23.52	23.44
Arch3		24.73	23.74	22.96	23.11	23.05
Arch4		22.02	21.79	21.65	21.18	21.26

masters. In TDMA2, two slots are allotted to the AVLink and USB and one slot for the remaining masters. In both the TDMA schemes, if a slot is not used by a master, then a secondary RR scheme is used to grant the slot to a master with a pending request. SP1 is a static priority scheme with the AVLink controller having a maximum priority followed by the USB, ARM926EJ-S, DMA, A/V encoder, and the A/V decoder. The priorities for the AVLink controller and USB are interchanged in SP2, with the other priorities remaining the same as in SP1.

For architecture *Arch1*, shown in Figure 22a, performance suffers because of frequent arbitration conflicts in the shared AHB bus. The shaded cells indicate scenarios where the bandwidth constraints for the USB and/or AVLink controller are not met. From Table II, we can see that none of the arbitration policies in *Arch1* satisfy the constraints.

To decrease arbitration conflicts, we shift the new components to a dedicated AHB bus, as shown in Figure 22b. An AHB/AHB bridge is used to interface with the main bus. We split MEM5 and attach one of the memories (MEM6) to the dedicated bus and also add an interface to the SDRAM controller ports from the new bus, so that data traffic from the new components does not load the main bus as frequently. Table II shows a performance improvement for *Arch2* as arbitration conflicts are reduced. With the exception of the RR scheme, bandwidth constraints are met with all the other arbitration policies. The TDMA2 scheme outperforms TDMA1, because of the reduced load on the main bus from the AVLink component, which results in inefficient RR distribution of its four slots in TDMA1. TDMA2 also outperforms the SP schemes because SP schemes result in much more arbitration delay for the low-priority masters (ARM CPU, DMA), whereas TDMA2 guarantees certain bandwidth even to these low-priority masters in every frame.

Next, to improve performance we allocate the A/V decoder and AVLink components to separate AHB buses, as shown in Figure 22c. From Table I we see that the performance for *Arch3* improves only slightly over *Arch2*. The reason for the small improvement in performance is because there is not a lot of conflict (or time overlap) between transactions issued by the A/V decoder and AVLink components. As such, separating these components eliminates those few conflicts that exist between them, improving performance only slightly.

Statistics gathered during simulation indicate that the A/V decoder frequently communicates with the ARM CPU and the DMA. Therefore, with the intention of improving performance even further, we allocate the high-bandwidth USB and AVLink controller components to separate AHB buses and bring the A/V decoder to the main bus. Figure 22d shows the modified architecture *Arch4*.

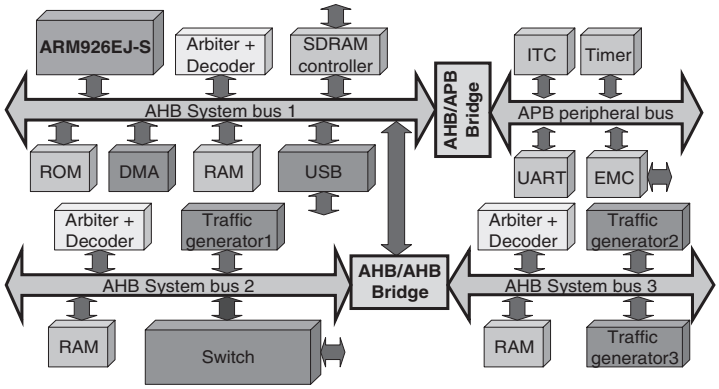


Fig. 23. SoC platform.

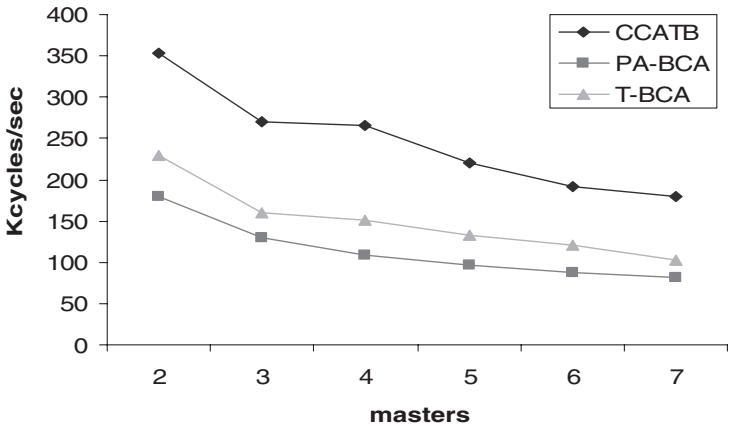
Performance figures from the table indicate that the SP1 scheme performs better than the rest of the schemes. This is because the SP scheme works well when requests from the high-bandwidth components are infrequent (since they have been allocated on separate buses). The TDMA schemes suffer because of several wasted slots for the USB and AVLink controller, which are inefficiently allocated by the secondary RR scheme.

We thus arrive at the *Arch4* topology together with the SP1 arbitration scheme as the best choice for the new version of the SoC design. We arrived at this choice after evaluating several other combinations of topology/arbitration schemes not shown here because of lack of space. It took us less than a day to evaluate these different communication design space points with our CCATB models and our results were verified by simulating the system with a more detailed PA-BCA model. It would have taken much longer to model and simulate the system with other approaches. The next section quantifies the gains in simulation speed and modeling effort for the CCATB modeling abstraction, when compared with other models.

8. SIMULATION AND MODELING EFFORT COMPARISON

We now present a comparison of the modeling effort and simulation performance for pin-accurate BCA (PA-BCA), transaction-based BCA (T-BCA), and our CCATB models. For the purpose of this study, we chose the SoC platform shown in Figure 23. This platform is similar to the one we used for exploration in the previous section, but is more generic and is not restricted to the multimedia domain. It is built around the AMBA 2.0 communication architecture and has an ARM926EJ-S processor ISS model with a test program running on it which initializes different components and then regulates data flow to and from the external interfaces, such as USB, switch, external memory controller, (EMC), and the SDRAM controller.

For the T-BCA model, we chose the approach used in [AHB CLI]. Our goal was to compare not only the simulation speeds, but also to determine how the speed changed with system complexity. We first compared speedup for a "lightweight" system comprising of just two traffic generator masters along with



2 TrafficGen2 + TrafficGen3
3 TrafficGen2 + TrafficGen3 + ARM
4 TrafficGen2 + TrafficGen3 + ARM + DMA
5 TrafficGen2 + TrafficGen3 + ARM + DMA + TrafficGen1
6 TrafficGen2 + TrafficGen3 + ARM + DMA + TrafficGen1 + USB
7 TrafficGen2 + TrafficGen3 + ARM + DMA + TrafficGen1 + USB + SWITCH

Fig. 24. Simulation speed comparison.

Table III. Comparison of Speedup and Modeling Effort

Model Abstraction	Average CCATB Speedup (x times)	Modeling Effort
CCATB	1	∼3 days
T-BCA	1.67	∼4 days
PA-BCA	2.2	∼1.5 wks

peripherals used by these masters, such as the RAM and the EMC. We gradually increased system complexity by adding more masters and their slave peripherals. Figure 24 shows the simulation speed comparison with increasing design complexity.

Note the steep drop in simulation speed when the third master was added—this is as a result of the detailed non-native SystemC model of the ARM926EJ-S processor, which considerably slowed down simulation. In contrast, the simulation speed was not affected as much when the DMA controller was added as the fourth master. This was because the DMA controller transferred data in multiple word bursts, which can be handled very efficiently by the transaction-based T-BCA and CCATB models. The CCATB particularly handles burst mode simulation very effectively and, consequently, has the least degradation in performance out of the three models. Subsequent steps added the USB switch and another traffic generator, which put considerable communication traffic and computation load on the system, resulting in a reduction in simulation speed. Overall, the CCATB abstraction level outperforms the other two models. Table III gives the average speedup of the CCATB over the PA-BCA and T-BCA models. We note that, on average, CCATB is faster than T-BCA by 67% and even faster than PA-BCA models by 120%.

Table IV. Comparison of Simulation Time during Exploration

Model Abstraction	Simulation Runs During Exploration	Total Simulation Time (in hours)
CCATB	37	63.8
T-BCA		113.1

As far as modeling accuracy is concerned, CCATB models—being variants of the transaction based BCA (T-BCA) approach—are just as accurate as T-BCA models, such as [AHB CLI], and, in some cases, even more accurate, such as when compared to the T-BCA models from [Zhu and Malik 2002], which do not capture sufficient detail, as discussed in Section 2. CCATB models are also just as accurate as PA-BCA models. This is demonstrated by the fact that the execution cycle counts for the CCATB, T-BCA, and PA-BCA models is the same for the case of the example we used for modeling speed and accuracy comparison in Figure 23. It might appear that CCATB suffers from a loss of accuracy when compared to PA-BCA models, but that is not the case, since the speedup obtained by CCATB over PA-BCA is as a result of sacrificing visibility of signals at every cycle boundary (as explained in Section 5), which has a significant simulation overhead. In addition, CCATB models use event synchronization semantics [Grötker et al. 2002] to handle intratransaction cycle level events (such as interrupts), thus capturing not only the system events occurring between different transactions, but also within transactions, which can affect accuracy and thus maintain the same level of accuracy as in PA-BCA models.

Table III also shows the time taken to model the communication architecture at the three different abstraction levels by a designer familiar with AMBA 2.0. While the time taken to capture the communication architecture and model the interfaces took just 3 days for the CCATB model, it took a day more for the transaction-based BCA, primarily due to the additional modeling effort to maintain accuracy at cycle boundaries for the bus system. It took almost 1.5 weeks to capture the PA-BCA model. Synchronizing and handling the numerous signals and design verification were the major contributors for the additional design effort in these models. CCATB models are thus faster to simulate and need less modeling effort compared to T-BCA and PA-BCA models.

To understand the implication of the speedup shown in Figure 24, on a typical communication architecture exploration effort, we considered the multimedia subsystem shown in Figure 21 and performed communication architecture exploration with greater test data volume and a correspondingly much larger test bench running on the ARM processor. We performed exploration for several different combinations of topology and communication parameters. Table IV shows the number of simulation runs during this exploration study and the overall simulation time for all the simulation runs, for the CCATB and the T-BCA models.

It can be seen that the exploration phase greatly benefits from the use of a faster CCATB simulation abstraction, which cuts down the simulation time by several days when compared to a similar exploration effort with the T-BCA model abstraction. With increasing communication architecture complexity, and ever-increasing levels of component integration in the SoCs of the future, we believe that the CCATB modeling abstraction, which is accurate, fast to

simulate, and takes less time to model, will be of immense use to system architects and designers.

9. CONCLUSION AND FUTURE WORK

Early exploration of system-on-chip communication architectures is extremely important to ensure efficient implementation and for meeting performance constraints. We presented the *Cycle Count Accurate at Transaction Boundaries* (CCATB) modeling abstraction, which is a fast, efficient, and flexible approach for exploring the vast communication space for shared-bus architectures in SoC designs. Our model enables plug-and-play exploration of various facets of the communication space, allowing master, slave, and bus IPs to be easily replaced with their architecture variants, and quickly estimating the impact on system performance. We also propose a five-layer modeling methodology that incorporates our CCATB abstraction level in a system design flow. Interface refinement from higher abstraction to lower levels in the design flow is simplified as we avoid altering the interface between IPs and the communication channel as much as possible. This also eases cosimulation of SoC IPs modeled at different abstraction levels in our system flow. We described the mechanisms responsible for speedup at the CCATB modeling abstraction, which enable fast and efficient exploration of the communication design space, early in the design flow. We have successfully applied our approach for exploring several industrial strength SoC subsystems. Two such exploration case studies from the broadband communication and multimedia domains, are presented in this paper. We also showed that the CCATB models are faster to simulate than pin-accurate BCA (PA-BCA) models by as much as 120%, on average, and are also faster than transaction-based BCA (T-BCA) models by 67%, on average. In addition, the CCATB models take less time to model than T-BCA and PA-BCA models. Our future work will focus on automatic refinement of CCATB models from high-level TLM models and interface refinement from CCATB down to the pin-accurate BCA abstraction level for RTL cosimulation purposes. We also intend to look at how the CCATB abstraction can speed up simulation for a multihop packet-switched network-on-chip (NoC) type of communication architecture.

REFERENCES

AHB CLI. 2003. http://www.arm.com/armtech/ahbcli.

AMBA AXI. http://www.arm.com/armtech/AXI.

BEN-ROMDHANE, M., MADISETTI, V., AND HINES, J. W. 1996. *Quick-Turnaround ASIC Design in VHDL: Core-Based Behavioral Synthesis*. Kluwer Academic Pub. Norwell, MA.

BERGAMASCHI, R. A. AND RAJE, S. 1996. Observable time windows: Verifying the results of high-level synthesis. In *European Conference on Design and Test*. 40–50.

CALDARI, M., CONTI, M., COPPOLA, M., CURABA, S., PIERALISI, L., AND TURCHETTI, C. 2003. Transaction-level models for AMBA bus architecture using SystemC 2.0 In *Design Automation and Test in Europe Conference and Exhibition*. 26–31.

COWARE. http://www.coware.com.

DAVIS, J. A. AND MEINDL, J. D. 1998. Is Interconnect the weak link? *Circuit and Device Magazine*, 30–36.

D&T ROUNDTABLE. 2001. System-on-chip specification and modeling using C++: Challenges and opportunities. *IEEE Design and Test of Computers, 18*, 3, 115–123.

FLYNN, D. 1997. AMBA: Enabling reusable on-chip designs. *IEEE Micro, 17*, 4.

GAJSKI, D. D., ZHU, J., DÖMER, R., GERSTLAUER, A., AND ZHAO, S. 2000. *SpecC: Specification Language and Methodology*. Kluwer Academic Publ. Norwell, MA.

GRÖTKER, T., LIAO, S., MARTIN, G., AND SWAN, S. 2002. *System Design with SystemC*. Kluwer Academic Publ. Norwell, MA.

HOFMANN, R. AND DRERUP, B. 2002. Next generation CoreConnect processor local bus architecture. In *Annual IEEE International ASIC/SOC Conference*. 25–28.

JANG, H., KANG, M., LEE, M., CHAE, K., LEE, K., AND SHIM, K. 2004. High-level system modeling and architecture exploration with SystemC on a network SoC: S3C2510 Case Study. In *Design Automation and Test in Europe Conference and Exhibition*. 538–543.

LOGHI, M., ANGIOLINI, F., BERTOZZI, D., BENINI, L., AND ZAFALON, R. 2004. Analyzing on-chip communication in a MPSoC environment. In *Design Automation and Test in Europe Conference and Exhibition*. 752–757.

NCSYSTEMC. http://www.cadence.com/products/functional_ver/nc-systemc/index.aspx.

NICOLESCU, G., SUNGJOO YOO, AND JERRAYA, A. A. 2001. Mixed-level cosimulation for fine gradual refinement of communication in SoC design. In *Design Automation and Test in Europe Conference and Exhibition*. 754–759.

OCP. http://www.ocpip.org.

OGAWA, O., BAYON DE NOYER, S., CHAUVET, P., SHINOHARA, K., WATANABE, Y., NIIZUMA, H., SASAKI, T., AND TAKAI, Y. 2003. A practical approach for bus architecture optimization at transaction level. In *Design Automation and Test in Europe Conference and Exhibition*. 176–181.

PASRICHA, S. 2002. Transaction level modeling of SoC with SystemC 2.0. In *Synopsys User Group Conference*. 55–59.

PASRICHA, S., DUTT, N., AND BEN-ROMDHANE, M. 2004a. Extending the transaction level modeling approach for fast communication architecture exploration. In *Design Automation Conference*, 113–118.

PASRICHA, S., DUTT, N., AND BEN-ROMDHANE, M. 2004b. Fast exploration of bus-based on-chip communication architectures. In *International Conference on Hardware/Software Codesign and System Synthesis*. 242–247.

PAULIN, P. G., PILKINGTON, C., AND BENSOUDANE, E. 2002. StepNP: A system-level exploration platform for network processors. *IEEE Design and Test of Computers, 19*, 6, 17–26.

ROWSON, J. A. AND SANGIOVANNI-VINCENTELLI, A. 1997. Interface-based design. In *Design Automation Conference*. 178–183.

SCANDURRA, A., FALCONERI, G., AND JEGO, B. 2003. STBus communication system: Concepts and sefinitions. *Reference Guide, STMicroelectronics.*

SÉMÉRIA, L. AND GHOSH, A. 2000. Methodology for hardware/software co-verification in C/C++. In *Asia and South Pacific Design Automation Conference*. 405–408.

SYLVESTER, D. AND KEUTZER, K. 1998. Getting to the bottom of deep submicron. In *IEEE/ACM International Conference on Computer-Aided Design*. 203–211.

SYSTEM STUDIO. http://www.synopsys.com/products/designware/system_studio/system_studio.html

WINGARD, D. 2001. MicroNetwork-based integration for SOCs. In *Design Automation Conference*. 673–677.

YIM, J., HWANG, Y., PARK, C., CHOI, H., YANG, W., OH, H., PARK, I. AND KYUNG, C. 1997. A C-based RTL design verification methodology for complex microprocessor. In *Design Automation Conference*. 83–88.

ZHU, X. AND MALIK, S. 2002. A hierarchical modeling framework for on-chip communication architectures. In *IEEE/ACM International Conference on Computer-Aided Design*. 663–670.

Received July 2004; revised April 2005 and June 2006; accepted December 2006