Rapid Exploration of Bus-based Communication Architectures at the CCATB Abstraction [*]

Sudeep Pasricha†, Nikil Dutt† and Mohamed Ben-Romdhane‡

†Center for Embedded Computer Systems
University of California Irvine
Irvine, CA 92697-3425, USA
1 (949) 824-2248
{sudeep, dutt}@cecs.uci.edu

‡Conexant Systems Inc.
4000 Mac Arthur Blvd
Newport Beach, CA 92660 USA
1 (949) 483-4600
m.benromdhane@conexant.com

CECS Technical Report #04-11
May, 2004

[*] This work was partially supported by grants from Conexant Systems Inc. and UC Micro (03-029)

Rapid Exploration of Bus-based Communication Architectures at the CCATB Abstraction [*]

Sudeep Pasricha†, Nikil Dutt† and Mohamed Ben-Romdhane‡

†Center for Embedded Computer Systems
University of California Irvine
Irvine, CA 92697-3425, USA
1 (949) 824-2248
{sudeep, dutt}@cecs.uci.edu

‡Conexant Systems Inc.
4000 Mac Arthur Blvd
Newport Beach, CA 92660 USA
1 (949) 483-4600
m.benromdhane@conexant.com

CECS Technical Report #04-11
May, 2004

Abstract

As a result of improvements in process technology, more and more components are being integrated into a single System-on-Chip (SoC) design. Communication between these components is increasingly dominating critical system paths and frequently becomes the source of performance bottlenecks. It therefore becomes extremely important for designers to explore the communication space early in the design flow. Traditionally, pin-accurate Bus Cycle Accurate (PA-BCA) models were used for exploring the communication space. To speed up simulation, transaction based Bus Cycle Accurate (T-BCA) models have been proposed, which borrow concepts found in the Transaction Level Modeling (TLM) domain. More recently, the Cycle Count Accurate at Transaction Boundaries (CCATB) modeling abstraction was introduced for fast communication space exploration. In this technical report, we describe the mechanisms that produce the speedup in CCATB models and demonstrate the effectiveness of the CCATB exploration approach with the aid of a case study involving an AMBA 2.0 based SoC subsystem used in the multimedia application domain. We also analyze how the achieved simulation speedup scales with design complexity and show that SoC designs modeled at the CCATB level simulate 120% faster than PA-BCA and 67% faster than T-BCA models on average.

[*] This work was partially supported by grants from Conexant Systems Inc. and UC Micro (03-029)

Contents

List of Figures

List of Tables

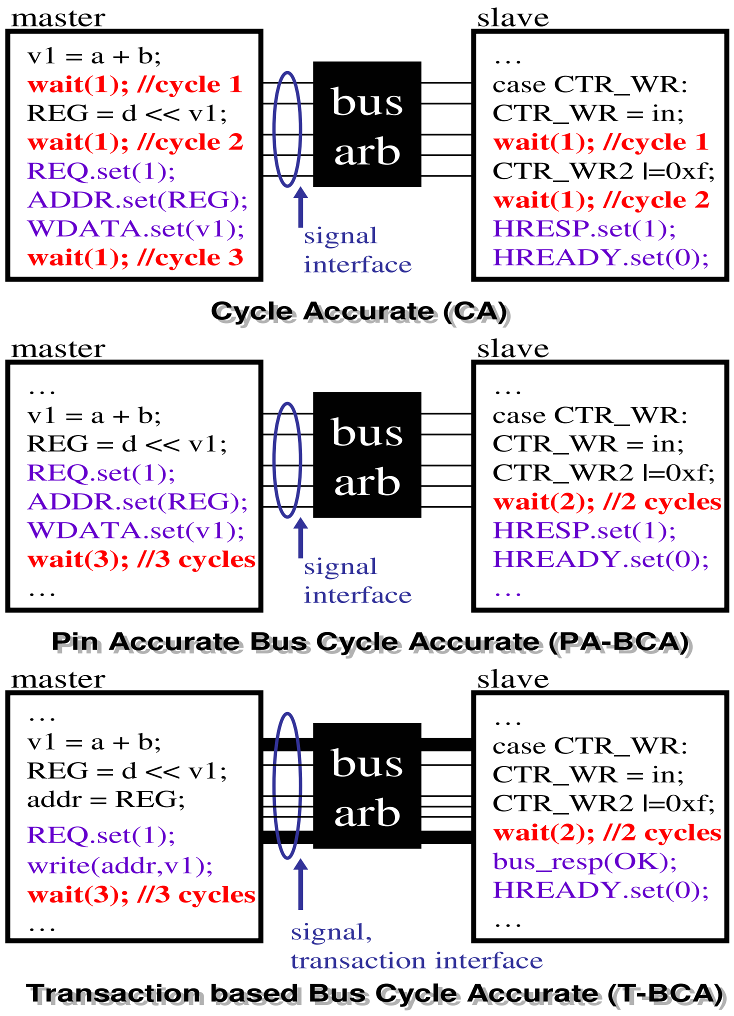
1. Introduction

Over the years, System-on-Chip (SoC) designs have evolved from fairly simple uni-processor, single-memory designs to massively complex multiprocessor systems with several on-chip memories, standard peripherals and ASIC blocks. As more and more components are integrated into these designs to share the ever increasing processing load, there is a corresponding increase in the communication between these components. Inter-component communication is often in the critical path of a SoC design and is a very common source of performance bottlenecks. It thus becomes imperative for system designers to focus on exploring the communication design space.

Shared-bus based communication architectures such as AMBA [1], CoreConnect [2], WishBone [3] and OCP [4] are 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 re-engineering effort due to the highly complex nature of these systems. To overcome these problems, designers have raised the modeling abstraction level above the RTL level. Figure 1 shows the frequently used modeling abstraction levels for communication space exploration, usually captured with high level languages such as C/C++ [5]. In Cycle Accurate (CA) models [6][18], 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. Bus Cycle Accurate (BCA) models [7] capture the system at a higher abstraction level than CA models. Components are modeled at a less detailed behavioral level, which allows rapid system prototyping and considerable simulation speed over RTL. The component interface and the bus however are still modeled at a cycle and signal accurate level, which enables accurate communication space exploration. However, with the increasing role of embedded software and rising design complexity, even the simulation speedup gained with BCA models is not enough.

Recent research efforts [11][12][13]14] have focused on using concepts found in the Transaction Level Modeling (TLM) [8][9][10] domain to speed up BCA model simulation. Transaction Level Models are very high level 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 signal-accuracy is omitted, these models are fast to simulate and are useful for early embedded software development and functional validation of the system [8]. Transaction based BCA (T-BCA) models [11][12][13][14] make use of the read/write function call interface, optionally with a few signals to maintain bus cycle accuracy. The simpler interface reduces modeling effort and the function call semantics result in faster simulation speeds.

More recently, we introduced the *Cycle Count Accurate at Transaction Boundaries* (*CCATB*) modeling abstraction [20][21] for fast exploration of communication architectures. CCATB extends the TLM modeling abstraction to speed up system prototyping and more importantly simulation performance, while maintaining cycle count accuracy during communication space exploration.
In this report we will describe the mechanisms behind the speedup obtained in CCATB models. We will present a simulation implementation of the CCATB modeling abstraction, for high performance shared bus architectures. To underline the effectiveness of our exploration approach, we will describe a case study involving an AMBA 2.0 based SoC subsystem used in the multimedia application domain. We will also compare simulation performance for CCATB, PA-BCA and T-BCA models and analyze the scalability of these approaches with design complexity.


master
slave
bus arb
signal interface
Cycle Accurate (CA)
master
slave
bus arb
signal interface
Pin Accurate Bus Cycle Accurate (PA-BCA)
master
slave
bus arb
signal, transaction interface
Transaction based Bus Cycle Accurate (T-BCA)

```
v1 = a + b;
wait(1); //cycle 1
REG = d << v1;
wait(1); //cycle 2
REQ.set(1);
ADDR.set(REG);
WDATA.set(v1);
wait(1); //cycle 3
```

```
...
case CTR_WR:
CTR_WR = in;
wait(1); //cycle 1
CTR_WR2 |=0xf;
wait(1); //cycle 2
HRESP.set(1);
HREADY.set(0);
```

```
...
v1 = a + b;
REG = d << v1;
REQ.set(1);
ADDR.set(REG);
WDATA.set(v1);
wait(3); //3 cycles
...
```

```
...
case CTR_WR:
CTR_WR = in;
CTR_WR2 |=0xf;
wait(2); //2 cycles
HRESP.set(1);
HREADY.set(0);
...
```

```
...
v1 = a + b;
REG = d << v1;
addr = REG;
REQ.set(1);
write(addr,v1);
wait(3); //3 cycles
...
```

```
...
case CTR_WR:
CTR_WR = in;
CTR_WR2 |=0xf;
wait(2); //2 cycles
bus_resp(OK);
HREADY.set(0);
...
```

Figure 1: Modeling Abstractions for Exploration

The report is organized as follows. Section 2 briefly discusses requirements for a communication design space exploration effort. Section 3 gives an overview of the CCATB modeling abstraction level for communication architecture exploration. Section 4 presents an implementation of the CCATB simulation model. Section 5 illustrates how our approach can be used for exploration with heterogeneous component model specifications frequently encountered in design efforts Section 6 describes a case study which uses CCATB models to explore the communication space of a multimedia SoC subsystem. Section 7 compares modeling effort and simulation speeds for the CCATB and BCA models, and shows how the speeds scale with increasing system complexity. Finally, Section 8 concludes the report and gives directions for future research.

2. Requirements for Communication Design Space Exploration

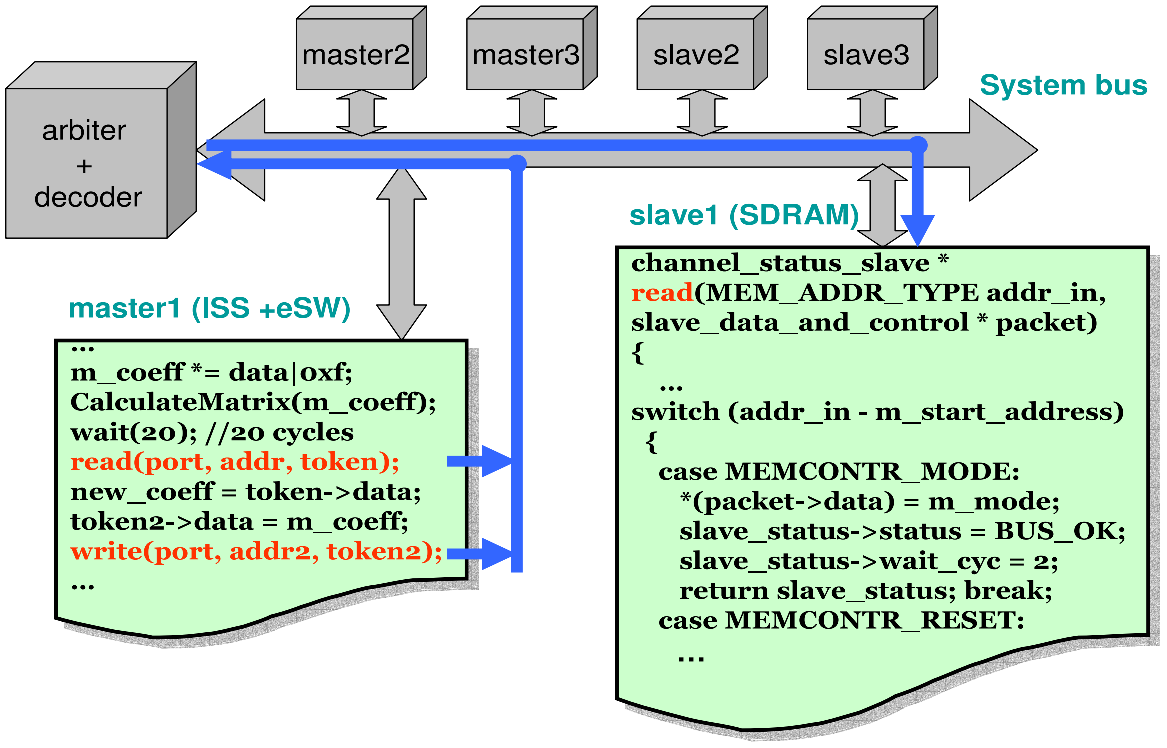
After system designers have performed hardware/software partitioning and architecture mapping in a typical design flow [10], they need to select a communication architecture for the design. The selection is complicated by the plethora of choices [1][2][3][4] that a designer is confronted with. Factors 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 [1] have several parameters which 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 study presented in this report, we use our approach to configure a communication architecture once the selection process is completed. Exploration studies focusing on the selection of appropriate communication architectures using our approach can be found in [20][21].

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 [19]. 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 [16], annotated with timing details and modeled at a granularity which would capture their precise functionality, yet not weigh down simulation speed due to unnecessary detail. Performance numbers would then be obtained by simulating the working of the entire system – including running embedded software on the CPU architecture model. 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.

3. CCATB Overview

To enable fast exploration of the communication design space, we introduced a novel modeling abstraction level called *Cycle Count Accurate at Transaction Boundaries* (*CCATB)* [20][21]. 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 [8] except that we additionally pass bus protocol specific control and timing information. Unlike 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 detailed cycle/pin accurate system model. A similar concept can be found in [15] 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 intra-transaction visibility to gain simulation speedup.

We chose SystemC 2.0 [8][9] to capture designs at the CCATB abstraction level, as it provides a rich set of primitives for system modeling. Busses in CCATB are modeled by extending the generic TLM *channel* [8] 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 [8]. Masters are active blocks with (possibly) several computation threads and ports to interface with busses. Figure 2 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 busses). *addr* is the address of the slave to send the transaction to. *token* is a structure that contains pointers to data and control information.



```
...
m_coeff *= data|oxf;
CalculateMatrix(m_coeff);
wait(20); //20 cycles
read(port, addr, token);
new_coeff = token->data;
token2->data = m_coeff;
write(port, addr2, token2);
...
```

```
channel_status_slave *
read(MEM_ADDR_TYPE addr_in,
slave_data_and_control * packet)
{
  ...
switch (addr_in - m_start_address)
 {
  case MEMCONTR_MODE:
    *(packet->data) = m_mode;
    slave_status->status = BUS_OK;
    slave_status->wait_cyc = 2;
    return slave_status; break;
  case MEMCONTR_RESET:
    ...
```

Figure 2: CCATB Transaction Example

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.

4. 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 [17]. OO transaction completion allows slaves to relinquish control of the bus, complete received transactions in any order and then request for re-arbitration so a response can be sent back to the master for the completed transaction. OO latency period refers to the number of cycles that elapse after the slave releases control of the bus and before it requests for re-arbitration.

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 the bus subsystem, waiting for selection by the arbiter. R_{act} is a set of read/write requests actively executing in the 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. Let A be a superset of the sets X for all *p* bus subsystems in the entire system.

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

Next we define τ to be a transaction request structure, which 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

status is defined to be 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 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 (i) the number of cycles before the next read/write request is issued by the master and (ii) 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.

On a positive clock edge, master computation threads are triggered and possibly issue read/write transactions, which in turn trigger the *GatherRequests* procedure (Figure 3) in the bus module. *GatherRequests* simply adds the transaction request to the set of pending requests R_{pend} for the subsystem.

```
procedure GatherRequests()
begin
    if request then
        $\tau \Leftarrow request$
        $\tau.wait\_cyc \Leftarrow 0$
        $\tau.oo\_cyc \Leftarrow 0$
        $\tau.ooflag \Leftarrow FALSE$
        $R_{pend} \Leftarrow R_{pend} \bigcup \tau$
end
```

Figure 3: *GatherRequests* procedure

On the negative clock edge, the *HandleBusRequests* procedure (Figure 4) in the bus module is triggered to handle the communication requests in the system. This procedure calls the *HandleCompletedRequests* procedure (Figure 5) for every subsystem to check if any executing requests in R_{act} have completed, in which case the 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 period.

```
procedure HandleBusRequests()
begin
  for each set $X \in A$ do
      HandleCompletedRequests($R_{pend}$, $R_{act}$, $R_{oo}$)
      $T \Leftarrow$ ArbitrateRequest($R_{pend}$, $R_{act}$)
      for each request $\tau \in T$ do
          if ( $\tau$.ooflag == TRUE) then
              $R_{act} \Leftarrow R_{act} \bigcup \tau$
          else
              status $\Leftarrow$ issue( $\tau$.port, $\tau$.addr, $\tau$ )
              UpdateDelaysAndSets(status, $\tau$, $R_{act}$, $R_{oo}$)
      $\psi \Leftarrow$ DetermineIncrementPeriod(A)
      for each set $X \in A$ do
          for each request $\tau \in R_{oo}$ do
              $\tau$.oo_cyc $\Leftarrow$ $\tau$.oo_cyc - $\psi$
          for each request $\tau \in R_{act}$ do
              $\tau$.wait_cyc $\Leftarrow$ $\tau$.wait_cyc - $\psi$
      for each value $\lambda \in M$ do
          $\lambda \Leftarrow \lambda - \psi$
      simulation_time $\Leftarrow$ simulation_time + $\psi$
end
```

Figure 4: ***HandleBusRequests*** **procedure**

```
procedure HandleCompletedRequests($R_{pend}$, $R_{act}$, $R_{oo}$)
begin
  $S_{pend} \Leftarrow$ null ; $S_{act} \Leftarrow$ null ; $S_{oo} \Leftarrow$ null ;
  for each request $\tau \in R_{act}$ do
      if ( $\tau$.wait_cyc == 0) then
          notify( $\tau$.master, $\tau$.status)
      else
          $S_{act} \Leftarrow S_{act} \bigcup \tau$
  for each request $\tau \in R_{oo}$ do
      if ( $\tau$.oo_cyc == 0) then
          $S_{pend} \Leftarrow S_{pend} \bigcup \tau$
      else
          $S_{oo} \Leftarrow S_{oo} \bigcup \tau$
  $R_{pend} \Leftarrow S_{pend}$ ; $R_{act} \Leftarrow S_{act}$ ; $R_{oo} \Leftarrow S_{oo}$ ;
end
```

Figure 5: ***HandleCompletedRequests*** **procedure**

Next, we arbitrate to select requests from the pending request set R_{pend} which will be granted access to the bus. The function *ArbitrateRequest* (Figure 6) performs the selection based on the arbitration policy selected for every bus. We assume that a call to the *ArbitrateOnPolicy* 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 [17], 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.

```
function ArbitrateRequest($R_{pend}$, $R_{act}$)
begin
  $T \Leftarrow null$
  for each independent channel $c \in$ subsystem $R_{pend}$ do
      $T \Leftarrow T \bigcup ArbitrateOnPolicy(c, R_{pend})$
  $R_{pend} \Leftarrow R_{pend} \setminus T$
  return $T$
end
```

Figure 6: *ArbitrateRequest* function

After the call to *ArbitrateRequest*, 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 *τ.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 7).

```
procedure UpdateDelaysAndSets(status, $\tau$, $R_{act}$, $R_{oo}$)
begin
  if (status.stat == OK) then
      $\tau$.status = OK
      if (status.oo == TRUE) then
          $\tau$.ooflag $\Leftarrow$ TRUE
          $\tau$.oo_cyc $\Leftarrow$ status.(oo_delay + slave_int_delay
                        + slave_comp_delay + bridge_delay)
                        + $\tau$.arb_delay
          $\tau$.wait_cyc $\Leftarrow$ $\tau$.(busy_delay + burst_length_delay
                        + ppl_delay + bridge_delay + arb_delay)
          $R_{oo} \Leftarrow R_{oo} \bigcup \tau$
      else
          $\tau$.wait_cyc $\Leftarrow$ status.(slave_int_delay
                         + slave_comp_delay + bridge_delay)
                         + $\tau$.(busy_delay + burst_length_delay
                         + ppl_delay + arb_delay)
          $R_{act} \Leftarrow R_{act} \bigcup \tau$
  else
      $\tau$.status = ERROR
      $\tau$.wait_cyc $\Leftarrow$ status.(slave_int_delay
                    + bridge_delay + error_delay)
                    + $\tau$.(busy_delay + burst_length_delay
                    + ppl_delay + arb_delay)

end
```

Figure 7: *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 *τ.oo_cyc* with the number of cycles to wait before applying for re-arbitration, and *τ.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 due to the error.

After returning from the *UpdateDelaysAndSets* procedure, we find the minimum number of cycles (ψ) before we need to invoke the *HandleBusRequests* procedure again, by calling the *DetermineIncrementPeriod* function (Figure 8).

```
function DetermineIncrementPeriod(A)
begin
  $\psi \Leftarrow inf$
    for each set $X \in A$ do
        for each set $R_{pend} \in X$ do
            if $R_{pend} \neq NULL$ then
                $\psi \Leftarrow 1$
                return $\psi$
        for each set $R_{act} \in X$ do
            for each request $\tau` \in R_{act}$ do
                $\psi \Leftarrow min\{ \psi, \tau`.wait\_cyc \}$
        for each set $R_{oo} \in X$ do
            for each request $\tau`` \in R_{oo}$ do
                $\psi \Leftarrow min\{ \psi, \tau``.oo\_cyc \}$
    for each value $\lambda \in M$ do
        $\psi \Leftarrow min\{ \psi, \lambda \}$
  return $\psi$
end
```

Figure 8: ***DetermineIncrementPeriod*** **function**

This function returns the minimum value out of the wait cycles for every executing request (τ.wait_cyc), out-of-order request cycles for all waiting OO requests (τ.oo_cyc) and the next request latency cycles for every master (λ). If there is a pending request which needs to be serviced in the next cycle, the function returns 1, which is the worst case return value. By default, the *HandleBusRequests* procedure is invoked at the negative edge of 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 invocation of procedures and 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 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.

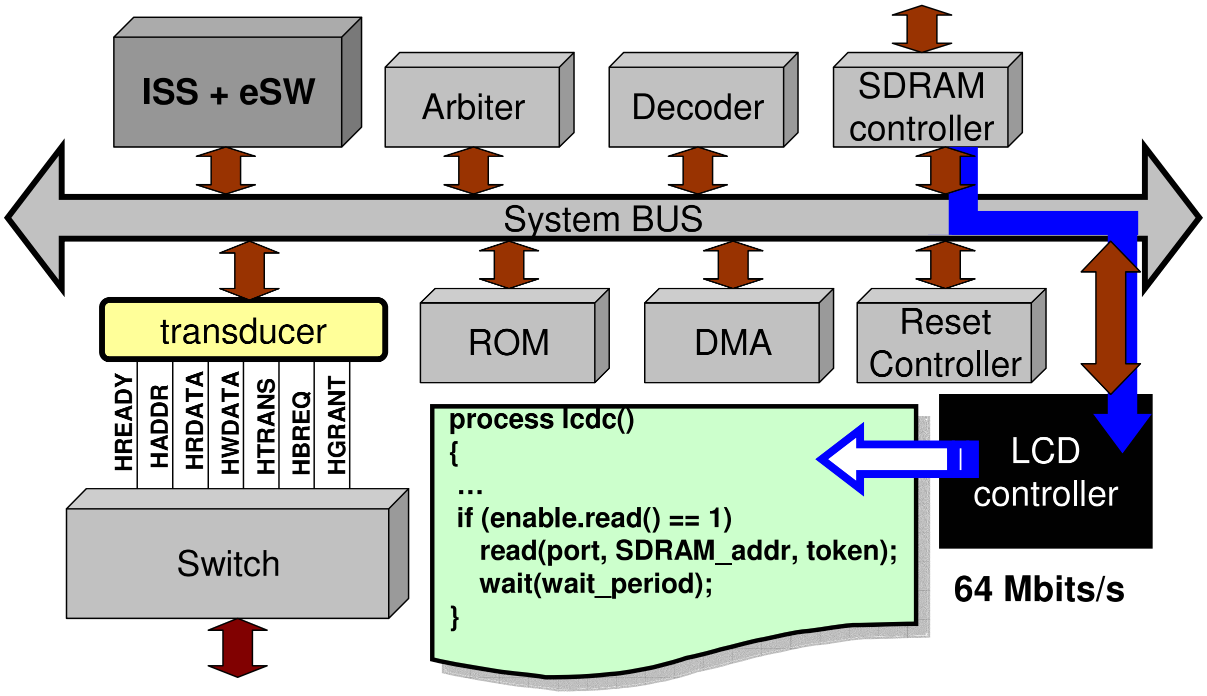
5. Exploration with Heterogeneous Component Specifications

Frequently, when the communication space exploration effort begins at the early stages of a SoC design flow, very little information is available about some components. Either the precise variant of the IP component to be used has yet to be decided, or the IP has still to be custom designed. In such situations, designers might have an idea of the rate at which the IP would generate or consume data.

Then even without a precise definition of the functionality and programmable interface, we can simulate and explore the communication space with the IP. Let γ be the bus size in bits, δ be the bus operation frequency, ζ be the data rate and η be the burst size (if applicable). Then a good estimate of the IP communication load on the system can be obtained by issuing a transaction every θ cycles, where

$$\theta = \eta * \gamma * \delta / \eta$$

Essentially we issue a read or write transaction on the bus and then wait for the wait period = θ – ε, where ε is the average number of cycles taken by the transaction. This value can be found out by an initial simulation run with the wait period set to the number of cycles specified by θ.



```
process lcdc()
{
 ...
 if (enable.read() == 1)
   read(port, SDRAM_addr, token);
   wait(wait_period);
}
```

Figure 9: Heterogeneous Component Simulation

Figure 9 shows the case of an LCD controller module which, once enabled, needs to receive data at the rate of 64 Megabits every second from memory for streaming multimedia display. It is connected to a 100 MHz bus with a data bus width of 32 bits. This implies that at most 32 bits can be transferred over the system bus in one cycle and the maximum theoretical bandwidth of the bus is 3.2 Gigabits/sec. Of course the actual bandwidth is much less than the theoretical bandwidth because of bus protocol, arbitration and IP computation delay overheads. Then according to our estimates, the LCD controller should load the bus with a read transaction every 50 cycles to maintain its throughput requirements. Since every transaction takes approximately 10 cycles on an average, the wait period is given by 50 – 10 = 40 cycles. We verified our approach by replacing the LCD controller IP with a more detailed behavioral model. The error for exploration parameters such as actual throughput, bus utilization and conflicts was less than 5%, which allowed us to get a fairly good idea whether the component throughput requirements were being met and also allowed us to determine the performance impact of introducing the component at a system level.

It is not uncommon to encounter the other extreme scenario – when components modeled at the detailed cycle accurate level and possibly having a pin accurate interface are available at the time of exploration early in the design flow. Instead of modeling the component at a behavioral level with the appropriate transaction interface, we can reuse the detailed model by introducing a transducer to translate pin values into transactions that can be understood by our system model. Figure 9 shows one such case where an AHB switch component with a pin accurate interface was integrated into our exploration system by quickly writing a transducer to allow it to communicate with our transaction based bus model. The additional component detail and the overhead of the transducer tend to slow down simulation speed, so it is preferable to capture components at a behavioral level whenever possible.

6. Exploration Case Study

To validate our modeling approach with the CCATB abstraction, we performed an exploration study with a consumer multimedia SoC subsystem which performs audio and video encoding for popular codecs such as MPEG. Figure 10 shows this platform, which is built around the AMBA 2.0 communication architecture [1], with a high performance bus (AHB or Advanced high performance bus) and a peripheral bus (APB or Advanced peripheral bus) for high latency, low bandwidth peripheral devices. The system has an ARM926EJ-S processor 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.

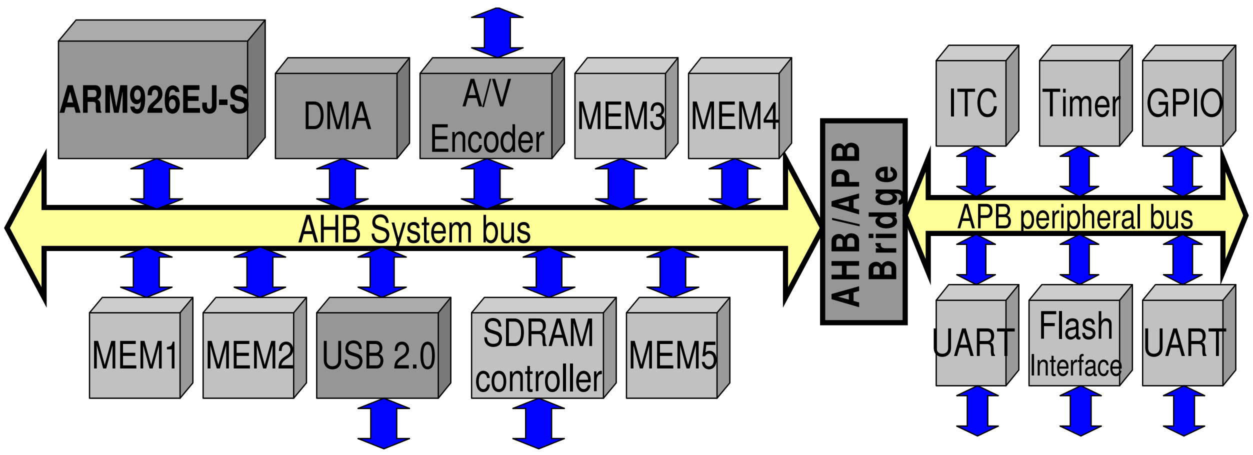


Figure 10: SoC Multimedia Subsystem

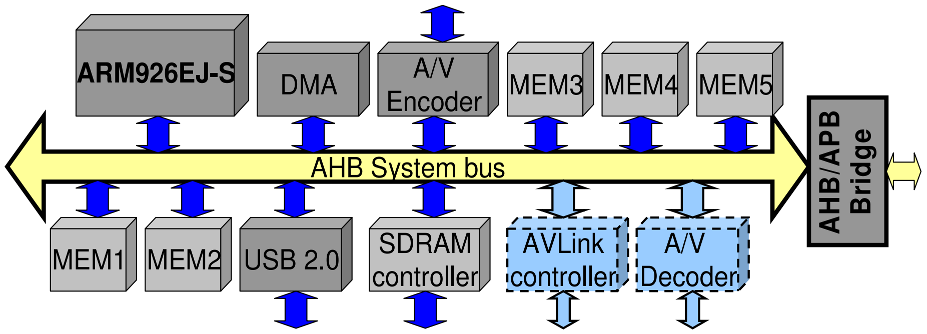
Consider a scenario where the designer wishes to extend the functionality of the encoder system to add support for audio/video decoding and an additional AVLink interface for streaming data. The final architecture must also meet peak bandwidth constraints for the USB component (480Mbps) and the AVLink controller interface (768Mbps). Figure 11(a) 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 till all the constraints were satisfied.

Table 1 shows the system performance (total cycle count for test program execution) for some of the architectures we considered, shown in Figure 11 (a), (b), (c) and (d). 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 4 slots are allotted to the AVLink controller, 2 slots to the USB, and 1 slot for the remaining masters. In TDMA2, 2 slots are allotted to the AVLink and USB, and 1 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, ARM926, 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.

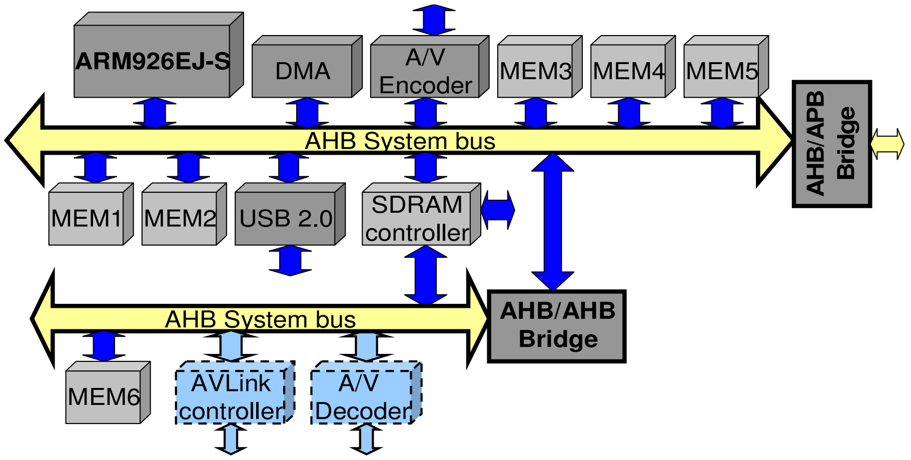
Architecture	**Arbitration Scheme**				
	RR	**TDMA1**	**TDMA2**	**SP1**	**SP2**
Arch1	27.24	24.65	25.06	25.72	26.49
Arch2	24.98	23.86	23.03	23.52	23.44
Arch3	24.73	23.74	22.96	23.11	23.05
Arch4	22.02	21.79	21.65	21.18	21.26

Table 1: Execution cycle counts (in millions of cycles)

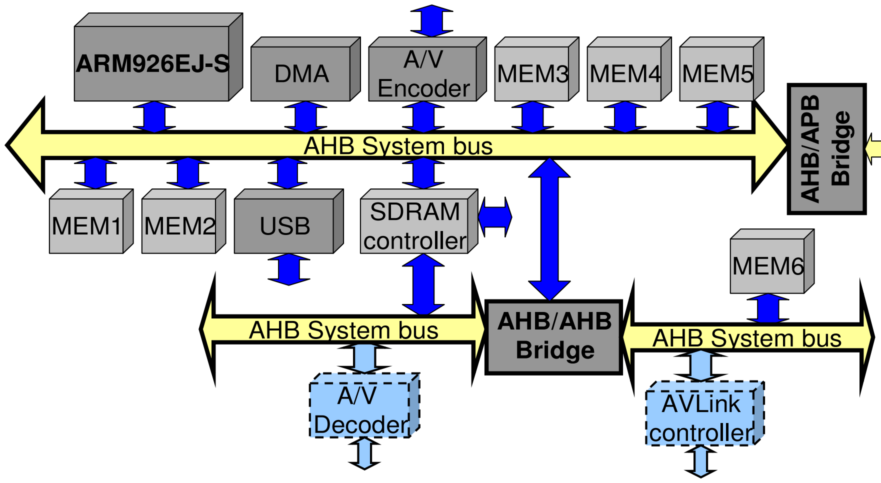
For architecture *Arch1*, performance suffers due to frequent arbitration conflicts in the shared AHB bus. The shaded cells indicate scenarios where the bandwidth constraints for the USB and/or AVLink controller are not met. From Table 1 we can see that none of the arbitration policies in *Arch1* satisfy the constraints.



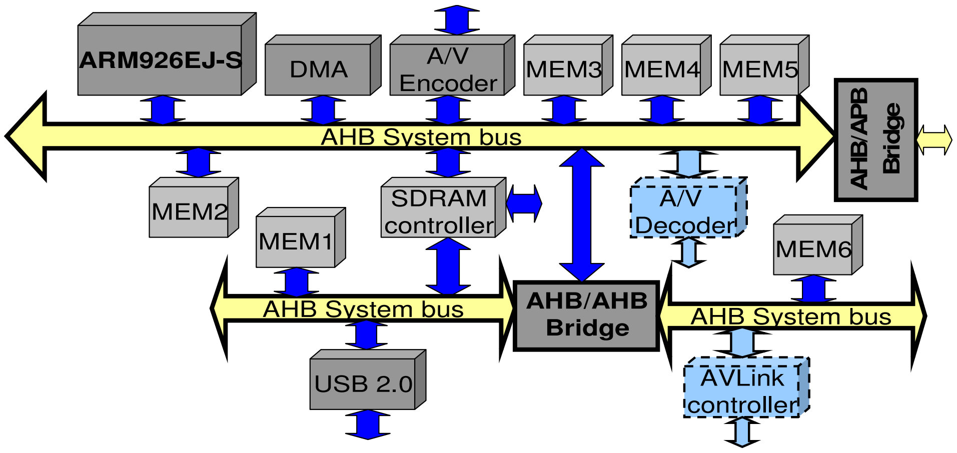
(a) Arch1



(b) Arch2



(c) Arch3



(d) Arch4

Figure 11: SoC Communication Architecture Topologies

To decrease arbitration conflicts, we shift the new components to a dedicated AHB bus as shown in Figure 11 (b). 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 1 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 4 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 busses, as shown in Figure 11 (c). From Table 1 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 busses, and bring the A/V decoder to the main bus. Figure 11(d) shows the modified architecture *Arch4*. 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 busses). 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 due to 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 pin accurate 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.

7. 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 12. 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 ARM926 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.

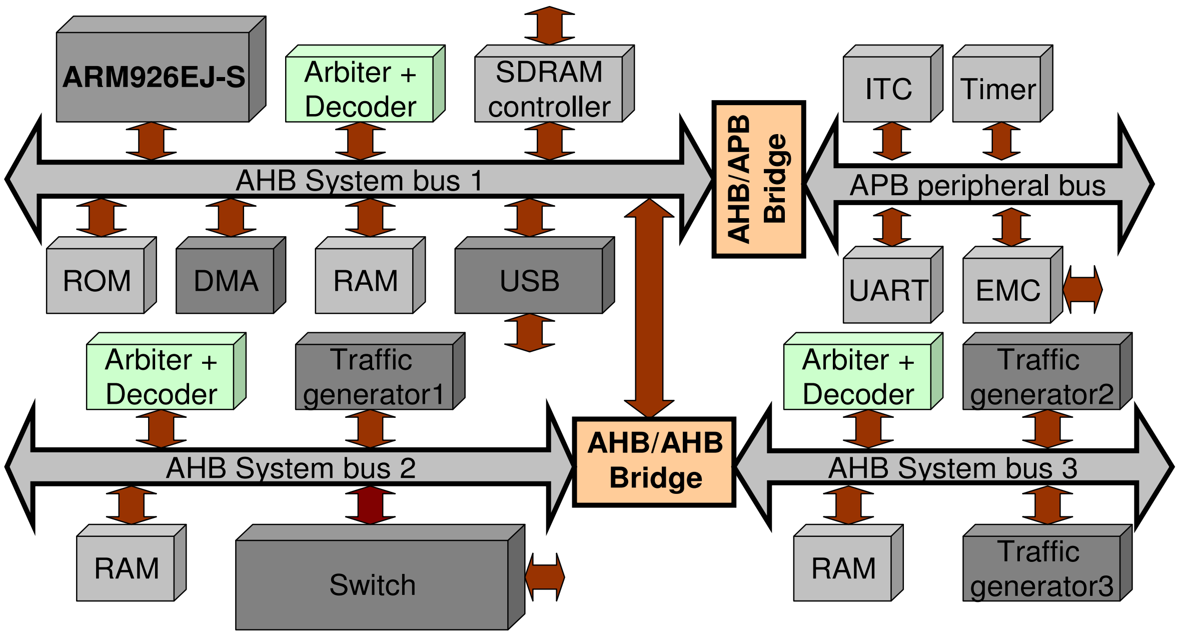


Figure 12: SoC platform

For the T-BCA model we chose the approach from [14]. Our goal was to compare not only the simulation speeds but also to ascertain how the speed changed with system complexity. We first compared speedup for a 'lightweight' system comprising of just 2 traffic generator masters along with 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 13 shows the simulation speed comparison with increasing design complexity.

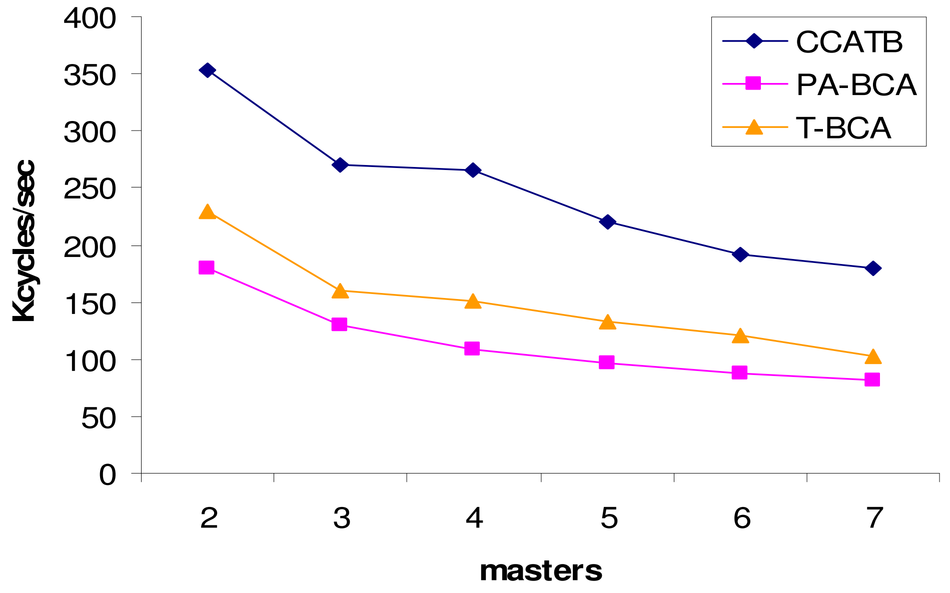


Figure 13: Simulation Speed Comparison

Note the steep drop in simulation speed when the third master was added – this is due to the detailed non-native SystemC model of the ARM926 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 2 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%.

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**

Table 2: Comparison of speed and modeling effort

Table 2 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. In summary, CCATB models are faster to simulate and need less modeling effort compared to T-BCA and PA-BCA models.

8. Conclusion

Early exploration of System-on-chip communication architectures is extremely important to ensure efficient implementation and for meeting performance constraints. We described the mechanisms responsible for speedup in our recently proposed CCATB modeling abstraction, which enable fast and efficient exploration of the communication design space, early in the design flow. We demonstrated the usefulness of our approach in a case study involving exploration of a multimedia SoC subsystem. Using models at the CCATB abstraction, we were able to quickly explore the impact of changes in the system and arrive at an architecture which met component bandwidth constraints and outperformed other choices. 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 co-simulation purposes.

9. References

[1] Flynn. “AMBA: enabling reusable on-chip designs”. *IEEE Micro*, *1997.*

[2] IBM CoreConnect http://www.chips.ibm.com/products/power pc/cores

[3] Wishbone Specification http://www.silicore.net/wishbone.htm

[4] Open Core Protocol International Partnership (OCP-IP). OCP datasheet, http://www.ocpip.org

[5] “System-on-Chip Specification and Modeling Using C++: Challenges and Opportunities”, *IEEE Design and Test May/June 2001*

[6] Joon-Seo Yim et al. “A C-Based RTL Design Verification Methodology for Complex Microprocessor”, *DAC, 1997*

[7] Luc Séméria, Abhijit Ghosh, "Methodology for Hardware/ Software Co-verification in C/C++", *ASP-DAC 2000*

[8] Sudeep Pasricha, “Transaction Level Modeling of SoC with SystemC 2.0”, in *Synopsys User Group Conference (SNUG)*, *2002*

[9] T. Grötker, S. Liao, G. Martin, S. Swan. “System Design with SystemC”. *Kluwer Academic Publishers, 2002.*

[10] D. Gajski et al., “*SpecC*: Specification Language and Methodology”, *Kluwer Academic Publishers*, *January 2000*

[11] Xinping Zhu , Sharad Malik, “A hierarchical modeling framework for on-chip communication architectures”, *IEEE/ACM International Conference on Computer-Aided Design, 2002*

[12] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, C. Turchetti “Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0”, *DATE 2003*

[13] O. Ogawa et al. “A Practical Approach for Bus Architecture Optimization at Transaction Level”, *DATE 2003*

[14] AHB CLI Specification http://www.arm.com/armtech/ahbcli

[15] Reinaldo A. Bergamaschi and Salil Raje, “Observable Time Windows: Verifying the Results of High-Level Synthesis”, *European Conference on Design and Test, 1996*

[16] Mohamed Ben-Romdhane et al. “Quick-Turnaround ASIC Design in VHDL: Core-Based Behavioral Synthesis” *Kluwer Academic Publishers, June 1996*

[17] AMBA AXI Specification http://www.arm.com/armtech/AXI

[18] H. Jang et al., “High-Level System Modeling and Architecture Exloration with SystemC on a Network SoC: S3C2510 Case Study”, *DATE 2004*

[19] M. Loghi et al. “Analyzing On-Chip Communication in a MPSoC Environment”, *DATE 2004*

[20] Sudeep Pasricha, Nikil Dutt, Mohamed Ben-Romdhane, “Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration", *Design and Automation Conference (DAC 2004), San Diego, CA, June 2004*

[21] Sudeep Pasricha, Nikil Dutt, Mohamed Ben-Romdhane, "High Level Design Space Exploration of Shared Bus Communication Architectures", *CECS Technical Report 04-06, March, 2004*