

# Performance Analysis of Systems With Multi-Channel Communication Architectures

Kanishka Lahiri  
Dept. of ECE  
UC San Diego  
klahiri@ece.ucsd.edu

Anand Raghunathan  
NEC USA C&C Research Labs  
Princeton, NJ  
anand@cctl.nj.nec.com

Sujit Dey  
Dept. of ECE  
UC San Diego  
dey@ece.ucsd.edu

## Abstract

This paper presents a novel system performance analysis technique to support the design of custom communication architectures for System-on-Chip ICs. Our technique fills a gap in existing techniques for system-level performance analysis, which are either too slow to use in an iterative communication architecture design framework (*e.g.*, simulation of the complete system), or are not accurate enough to drive the design of the communication architecture (*e.g.*, techniques that perform a “static” analysis of the system performance).

Our technique is based on a hybrid, trace-based performance analysis methodology where an initial co-simulation of the system is performed with the communication described in an abstract manner (*e.g.*, as events or abstract data transfers). An abstract set of traces are extracted from the initial co-simulation that contain necessary and sufficient information about the computations and communications of the system components.

The system designer then specifies a communication architecture by selecting a topology consisting of dedicated as well as shared communication channels (shared buses) interconnected by bridges, mapping the abstract communications to paths in the communication architecture, and finally customizing the protocol used for each channel. The traces extracted in the initial step are represented as a *Communication Analysis Graph* (CAG), and an analysis of the CAG provides an estimate of the system performance, as well as various statistics about the components and their communication. Experimental results indicate that our performance analysis technique achieves accuracy comparable to complete system simulation (an average error of 1.91%), while being over two orders of magnitude faster.

## 1 Introduction

The evolution of the System-on-Chip paradigm in electronic system design has the potential to offer the designer several benefits, including improvements in system cost, size, performance, power dissipation, and design turn-around-time. The ability to realize this potential depends on how well the designer exploits the configurability and customizability offered by the system-on-chip approach. Unfortunately, due to the increasing scale and complexity of electronic systems, the process of refining an abstract system specification into a system architecture that is optimized for the target application or domain is becoming an increasingly difficult task.

Achieving the disparate goals of reduction in design turn-around-time while better exploring system-level tradeoffs requires efficient and accurate analysis tools that guide the designer in the initial stages of the system design process. After completing an abstract specification of the system’s behavior, two important steps that need to be performed in order to derive a system architecture are partitioning/mapping and communication refinement. The

first step refers to the partitioning and mapping of parts of the system functionality into software that will execute on programmable processor(s), parts that will be implemented by re-using (possibly adapting) existing function-specific cores, and parts that will be compiled into hardware using hardware synthesis tools. The various components selected/assembled in the above step will require to share data and communicate in order to implement the system functionality. Hence the communication requirements of the system need to be refined into a communication architecture. Both of the above steps can significantly influence the quality of the resulting system architecture.

Researchers have worked on developing fast and accurate analysis techniques for various metrics such as performance, power, system cost, *etc.* for guiding the partitioning/mapping step [1, 2, 3, 4, 5, 6]. In this work, our focus is system performance analysis to drive the design of the communication architecture. System-level performance analysis techniques which consider the effects of the communication architecture can be broadly divided into the following categories:

- Approaches based on simulation of the entire system using models of the components and their communication at different levels of abstraction [7, 8]. The use of communication abstraction provides for a tradeoff between simulation time and accuracy, however, these techniques still require a simulation of the complete system.
- Static system performance estimation techniques that include models of the communication time [9, 10, 11, 12, 13]. These techniques often assume systems where the computations and communications can be statically scheduled. Further, the communication time estimates used in these systems are either overly optimistic, since they ignore dynamic effects such as wait time due to bus contention (*e.g.* [12, 13]), or are overly pessimistic by assuming a worst-case scenario for bus contention (*e.g.* [9]).

In this paper, we present a fast and accurate system performance analysis technique for driving communication architecture design. The relative accuracy and efficiency of our technique with respect

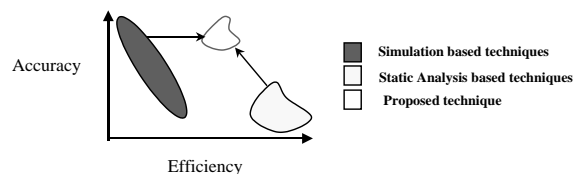


Figure 1: Accuracy and efficiency of the proposed system performance analysis technique relative to simulation based and static analysis based approaches

to simulation-based approaches and static performance analysis approaches is depicted in Figure 1. Our technique fills a gap in existing techniques for system-level performance analysis, which are either too slow to use in an iterative communication architecture design framework (e.g., simulation of the complete system), or are not accurate enough to drive the design of the communication architecture (e.g., techniques that perform a “static” analysis of the system performance).

Our technique is widely applicable since it supports a general communication architecture that can be based on shared or dedicated communication channels, while allowing the designer to specify a customized protocol for each communication channel. An important feature of our approach is its ability to model various dynamic effects of the communication architecture (such as wait times due to bus contention/conflicts), and to consider the inter-dependencies between the computations, synchronizations, and data communications performed by the various components while estimating the system performance.

Our technique is based on a hybrid, trace-based performance analysis methodology where an initial co-simulation of the system is performed with the communication described in an abstract manner (e.g., as events or abstract data transfers). An abstract set of traces are extracted from the initial co-simulation that contain necessary and sufficient information about the computations and communications of the system components. The system designer then specifies a communication architecture by selecting a topology consisting of dedicated as well as shared communication channels (shared buses) interconnected by bridges, mapping the abstract communications to paths in the communication architecture, and finally customizing the protocol used for each channel. The traces extracted in the initial step are represented as a *Communication Analysis Graph* (CAG), and an analysis of the CAG provides an estimate of the system performance, as well as various statistics about the components and their communication. Our initial work on analysis of a system containing a single shared bus is presented in [?]. The techniques presented in this paper enable are *significantly more general* since they enable us to study communication architectures that (a) consist of an arbitrary interconnected network of dedicated and shared communication channels, (b) have an arbitrary mapping of abstract communications to paths in the communication architecture (as we will see in the following sections, utilizing the flexibility offered by a general communication architecture template is critical in order to obtain a high-performance system implementation).

The basic idea of collecting an execution trace and using it for performance estimation has been used in the field of high-performance processor design, e.g., for cache simulation [14, 15]. We believe that our approach is the first to use this idea in the context of application-specific system performance analysis. Also, a key difference in our context is that in order to obtain an advantage over existing techniques (such as system simulation and static performance analysis) it is critical to abstract out information that is only necessary and sufficient from the initial co-simulation. This helps us to maintain accuracy comparable to complete system simulation on the one hand, while also achieving high efficiency and avoiding the problem of explosion of trace sizes. Since our analysis is trace based, the results provided by our tool are specific to the inputs used for the initial co-simulation. As in the case of any simulation-based analysis, this places additional responsibility on the system designer to provide meaningful stimuli. However, given that co-simulation is the most popular system-level analysis technique in practice, we feel that this additional burden on the designer is quite reasonable.

## 2 Motivation

In this section, we motivate the need for analysis techniques such as the ones presented in this paper. We demonstrate that the selection of a communication architecture that is well-suited to the communication traffic profiles of the application can have a signif-

icant impact on the system performance. We model the problem of selecting a communication architecture as consisting of three steps (i) the task of defining a *communication topology* that consists of a network of dedicated communication channels and shared communication channels, possibly connected by *bridges*, (ii) the task of mapping the abstract communication events onto the implementation architecture, and (iii) the task of selecting or customizing the protocol for each channel. We first illustrate these steps, and their effects on system performance, through examples.

Consider the system described in Figure 2(a), that consists of a set of four components that synchronize with each other, and access data in two shared memories. The figure provides a “behavioral” view of the communication, using a separate arc for each synchronization<sup>1</sup> (represented by dotted arcs) and data transfer (represented by solid arcs). Figures 2(b)-(d) show three alternative topologies for the system communication architecture. In Figure 2(b), a single shared bus is used to implement all data transfers, whereas in Figures 2(c) and 2(d), two buses (connected by a bridge) are used.

In addition to the structure or topology of the communication architecture, the mapping of abstract communication to the architecture is also different for the three architectures. In Figure 2(b), all data transfers to and from the shared memories are mapped to the only bus, while the inter-component synchronization is implemented using dedicated communication channels. In Figure 2(c), the data transfers between components  $C_1$  and  $C_2$ , and  $Memory1$ , are mapped to *Bus1*, while those between components  $C_3$  and  $C_4$ , and  $Memory2$ , are mapped to *Bus2*. In addition, a component can also use the bridge to communicate with components not attached directly to the same bus. For example, the data transfers between  $C_1$  or  $C_2$  and  $Memory2$  (similarly, the data transfers between  $C_3$  and  $C_4$  and  $Memory1$ ) will require to be performed through the bridge. In Figure 2(d), the mapping of communication events to buses is changed: components  $C_1$ ,  $C_3$  and  $Memory1$  now share a bus while the remaining components share the second bus.

There are several factors that need to be considered when determining which communication architecture among those of Figures 2(b)-(d) will result in the best overall system performance. These include:

- An architecture with multiple communication channels may allow for greater parallelism, because each of the communication channels can operate in parallel. As in the case of any other shared system resource, a shared bus limits the parallel communication between the different components that use it. However, as shown below, this does not necessarily imply that a shared bus will lead to poorer performance.
- Communicating with a component across multiple buses (through a bridge) can result in some additional overheads when compared to communicating with a component connected to the same bus. For example, in Figure 2(c), when component  $C_1$  needs to access  $Memory2$ , it first makes a request to the *Arbiter1* for accessing *Bus1*. When it receives a grant to use *Bus1* (after some handshaking time and a possible wait time due to contention for *Bus1*), the bridge is activated, and in turn makes a request to *Arbiter2*. Once *Arbiter2* grants its request, data transfer takes place at a rate determined by the minimum of the bandwidth of *Bus1* and the bandwidth of *Bus2*. Thus, multiple levels of handshaking and wait times due to contention may be involved, in addition to a potentially slower data transfer rate.
- The communication profiles of the system greatly influence the choice of communication architecture in several ways. For example, the concurrency (or lack of it) in the communication requirements of the various components will determine which

<sup>1</sup>Synchronization refers to communication without a transfer of data values [11]. It can be used to impose a desired structure on the control-flow of communicating parallel processes.

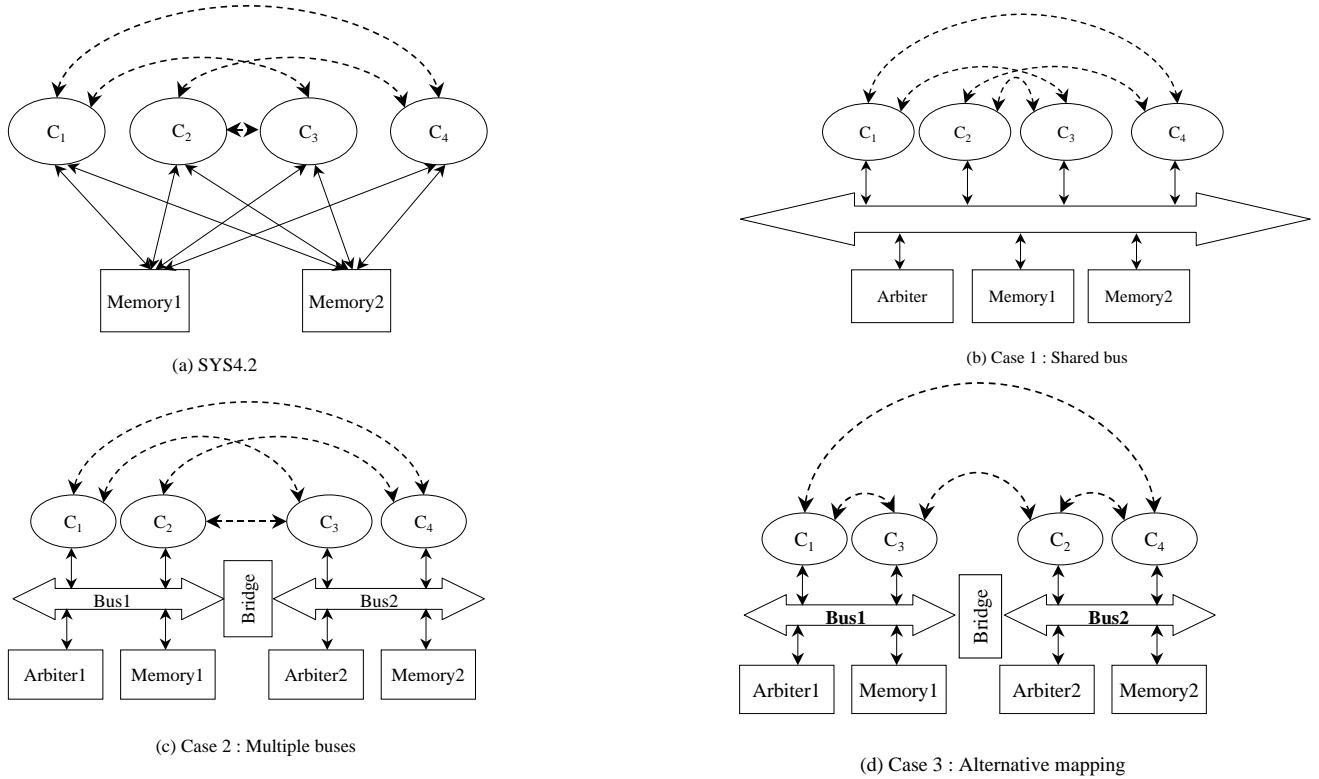


Figure 2: (a) An example system SYS4.2 with abstract communication and (b)-(d) Three alternative communication architectures

components are good candidates to share the same communication channel. Components with communication requirements that are largely exclusive in time are better candidates to share a bus since the likelihood of bus conflicts is lower.

While each of the above factors in itself has a significant influence on the design of the communication architecture, it is important to note that the factors also interact leading to various trade-offs. For example, consideration of the potential parallelism when using multiple buses suggests that the architectures of Figure 2(c) and 2(d) are superior to the architecture of Figure 2(b). In order to verify whether the above hypothesis holds, we used our tool to perform an analysis of the system performance for a long execution trace. The performance of the architectures of Figure 2(b)-(d) were 224031 cycles, 244002 cycles, and 165987 cycles, respectively. Thus, the single bus architecture of Figure 2(b) is actually 9% *faster than* the two bus architecture of Figure 2(c), contrary to the hypothesis. Upon careful analysis, we were able to explain the result as follows. Components  $C_1$  and  $C_3$  access *Memory1* more frequently, while  $C_2$  and  $C_4$  perform frequent accesses to *Memory2*. Under the communication architecture of Figure 2(c),  $C_3$  would have to go through the bridge in order to read or write data in *Memory2* (a similar argument holds for component  $C_3$  and *Memory1*). The incumbent overhead (due to the two levels of handshaking necessary) outweighs the potential improvements possible due to parallelism.

The communication architecture of Figure 2(d) avoids this problem. In addition, the communication profile of the system is such that communications of components  $C_1$  and  $C_2$  are often concurrent, while those of components  $C_3$  and  $C_4$  are also concurrent (this is due to the control-flow structure imposed by the inter-component synchronization). As a result, the communication architecture of Figure 2(d) also results in fewer bus conflicts for each bus compared to the architecture of Figure 2(c).

Exploring the above tradeoffs clearly requires an accurate quantitative analysis tool such as the one described in this paper. It is important that the analysis tool be general in terms of the communication architectures it can handle (*e.g.*, it is necessary to consider architectures containing multiple shared and dedicated channels, and not just single shared bus architectures as in [?]). Another important point to be noted is that the computation, synchronization, and communication times of various components in a system are *inter-dependent*. Thus, a separate analysis of the communication time alone will not necessarily reflect the total system performance accurately.

## 2.1 Effect of customizing bus protocols on system performance

Our next experiment shows that even after the communication topology has been selected, customizing the protocols and parameters of each channel can greatly influence the performance of the system. Consider a simple system shown in Figure 3 that consists of two components,  $C_1$  and  $C_2$ , that read and write to a global memory through a shared bus. In addition, the components synchronize with each other in order to ensure correct system operation. Each component makes requests to the arbiter which grants access to the shared bus. The system supports DMA mode transfers across the shared bus.

We performed several experiments to investigate the effect of the variation of DMA block size on the performance of the system. Here we present a test case, where the component  $C_1$  performs computations of average size 10 cycles and memory transfers of average size 100 bus words while  $C_2$  performs computations of average size 10 but memory transfers of average size only 10. Fig-

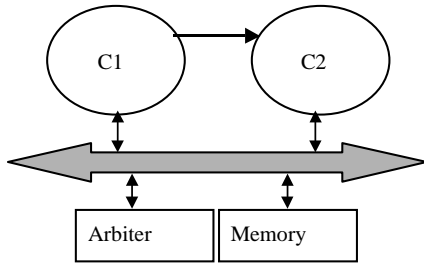


Figure 3: Example system to illustrate effect of DMA size on performance

Figure 4 shows the effect of varying DMA sizes (x-axis) on system performance (y-axis). We observe the following:

- The choice of bus parameters like DMA size can significantly affect system performance. For example Figure 4 shows the performance range for C2 for varying DMA sizes is 117-250 clock cycles.
- The optimal values of bus parameters like DMA block size depend heavily on the characteristics of the traffic seen on the bus. While increasing the DMA block size generally improves the performance of C2, it has a negative effect on C1, whose computation and bus access profile is different from that of C2.

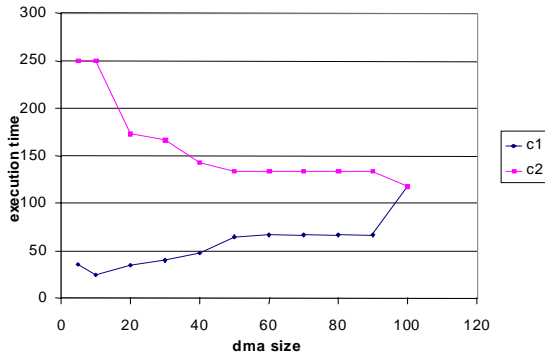


Figure 4: Effect of DMA size on performance

The above investigation demonstrates the criticality of selecting the optimal bus architectures and protocols, and thereby the need for fast and accurate performance analysis techniques that can evaluate the numerous possible bus architecture and parameter choices. As mentioned in Section 1, static analysis techniques for estimating the communication time in previous work are not accurate enough to drive the design of the communication architecture, while a complete HW/SW co-simulation of the system is too time consuming to perform iteratively in a design exploration framework.

### 3 Performance Analysis Methodology

In this section we describe the proposed hybrid two-phase methodology for fast and accurate system performance analysis that includes effects of the communication architecture. The complete methodology is shown in Figure 5.

The first phase of this methodology constitutes a pre-processing step in which system simulation of the HW/SW components is carried out, without considering the architecture that will implement

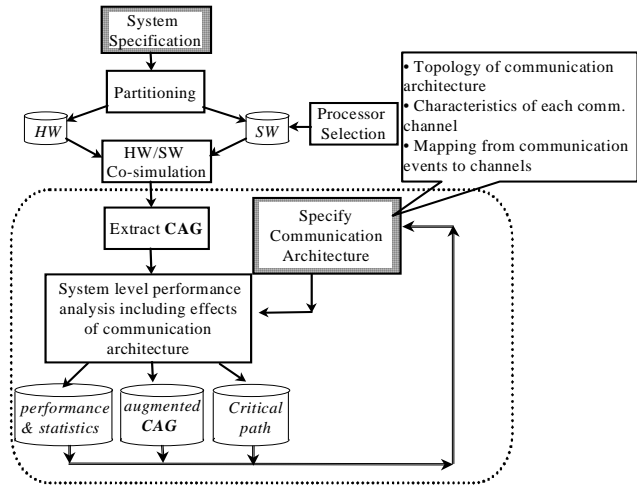


Figure 5: Two-phase performance analysis methodology

the communication. Here communication is modeled at the abstract level, by the exchange of events or tokens. The output of this step is a “timing inaccurate” system execution trace.

The second phase (enclosed within the dotted box in Figure 5) performs system performance analysis including the effects of the communication architecture. From the trace obtained in the first phase, we construct a *Communication Analysis Graph* (CAG), which captures the computations, communications, and the synchronizations seen during simulation of the entire system. In addition, the designer lays down the communication architecture to implement the communication events. Based on this architecture, the tool suitably manipulates the CAG and generates a “timing accurate” trace of the system performance under the given communication architecture. The output of our tool includes:

- An augmented version of the CAG, which has incorporated into it various latencies introduced by moving from an abstract communication model to an actual one.
- An estimate of the performance of the entire system.
- The system critical path
- Statistics regarding bus conflicts, the proportion of the system critical path occupied by each component, *etc.*

Using this two-phase methodology the time consuming pre-processing step needs to be performed only once. Alternative communication architectures can be evaluated within the second phase of the methodology. Being fast and accurate (as demonstrated in Section 4), the analysis tool can provide the designer with feedback regarding system performance under many alternative architectures within a short span of time.

As indicated in Figure 5, the second phase consists of three steps: abstracting information from the simulation trace and constructing the CAG, specifying the communication architecture, and analyzing the system performance under the given architecture. In the next three subsections we discuss each of these steps in turn.

#### 3.1 Extracting information from HW/SW co-simulation

Simulation traces obtained via HW/SW co-simulation can be very complex and detailed. Using these raw traces as a starting point for our analysis while desirable from an accuracy viewpoint, would result in high estimation times. Thus in our approach we extract only the information from the traces that is necessary and sufficient to perform accurate system performance analysis incorporating the effects of the communication architecture.

This extraction involves selective omission of unnecessary details regarding the computations and communications of the system components (*e.g.*, the values of internal variables, the values of the data communicated, *etc.*). Further, the extraction process groups contiguous bursts of computation and communication into abstract computation and communication clusters. For example, in the traces shown in Figure 6, *Component1* and *Component2* each execute computations, communication with memory (via data transfers), and communication between each other (via synchronizing events). However we do not extract the exact values communicated, or details of each and every computation.

These traces are then converted into a *Communication Analysis Graph* (CAG). Figure 7 shows a simple CAG which represents the traces of Figure 6. The graph has two kinds of nodes — *computation* and *communication* — and a set of directed edges representing timing dependencies. Dependencies could arise due to the sequential nature of each component (control-flow dependencies), or due to inter-component synchronization and communication.

The CAG is acyclic because it is constructed from simulation traces where all dependencies have been *unrolled in time*. It is compact compared to the simulation traces, because several simulation events may be collapsed into a single node of the graph.

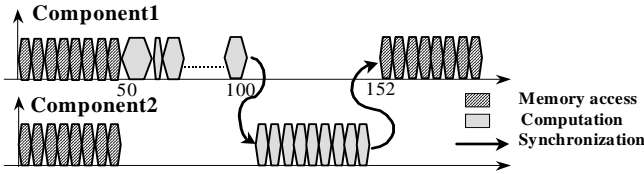


Figure 6: Traces generated by HW/SW co-simulation

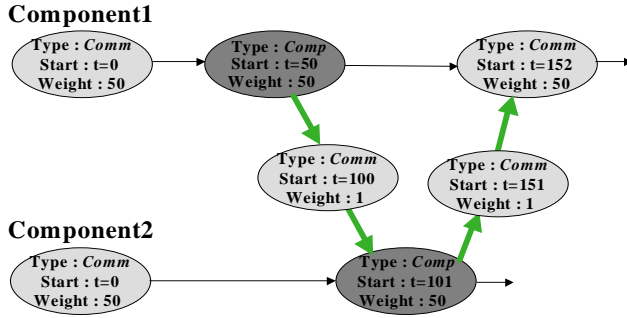


Figure 7: CAG generated from traces in Figure 6

### 3.2 Characteristics of the communication architecture

The analysis technique supports communication architectures with the following characteristics:

- On-chip communication using channels, where each channel may either be a dedicated link or a shared bus.
- Parameterized characterization of each channel.
- Arbitrary mapping of events to channels

We next briefly describe the characteristics that constitute a candidate communication architecture. Consider the system shown in Figure 8 consisting of 6 components  $c_1, \dots, c_6$  and a possible architecture consisting of two shared buses  $b_1$  and  $b_2$  and three dedicated links  $b_3, b_4, b_5$ . The communication architecture needs to be specified in terms of a parameterized characterization of each channel and as well as a mapping of communication events to channels.

Table 1: Description of communication channels

CHANNEL	PARAMETERS
$b_1$	$width = 32$ $speed = 66$ $dma-size = 256$ $latency = 10$
$b_2$	$width = 8$ $speed = 100$ $dma-size = 128$ $latency = 5$
...	...

Table 2: Mapping of components and events to channels

COMPONENT	ASSOCIATED CHANNELS AND EVENT MAPPING
$c_1$	$b_1$ : shared, priority=10; $b_5$ : dedicated event $e_1 \rightarrow b_1$ event $e_2 \rightarrow b_5$
$c_2$	$b_1$ : shared, priority=5; $b_3$ : dedicated $b_4$ : dedicated event $e_3 \rightarrow b_4$ event $e_4 \rightarrow b_1$
...	...

Table 1 shows a set of parameters for each channel. For example, for channel  $b_1$ ,  $width = 32$  (in bits),  $speed = 66MHz$ ,  $DMA\ transfer\ size = 256$  (in channel words) and  $latency = 10$  (in clock cycles) ( $latency$  represents the intrinsic overhead of setting up a communication over  $b_1$ ). In Table 2 the set of channels directly connected to each component is enumerated. For instance,  $c_1$  is connected to channels  $b_1$  and  $b_5$ . Additionally a channel such as  $b_5$ , which is dedicated to unidirectional communication from  $c_1$  to  $c_4$  is marked in Table 2 as a *dedicated link*. Since more than one component in Figure 8 may simultaneously try to use  $b_1$  for communication (memory is counted as a component, though one which does not initiate any communication by itself),  $b_1$  is indicated to be a *shared bus*. In case of a shared bus, a static priority (used to resolve bus contention) is defined for the component. For example, the priority of  $c_1$  on  $b_1$  is defined to be 10. Table 2 also defines a mapping from communication events to channels. Component  $c_1$  is associated with more than one channel, therefore each communication event (identified by a unique integer  $e_i$ ) that is generated by  $c_1$ , must be mapped to one of the available channels. For example,

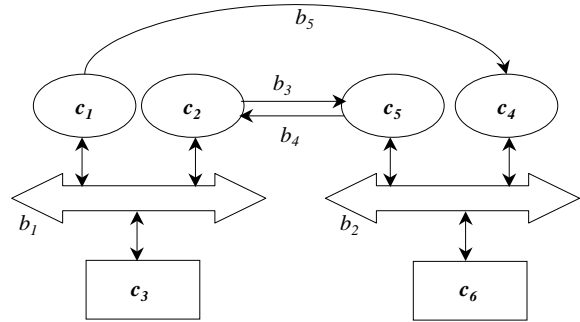


Figure 8: A communication architecture involving shared buses and dedicated links

the designer may choose to map a synchronizing event  $e_1$  emitted by  $c_1$  to the dedicated channel  $b_5$  and a data transfer  $e_2$  to the bus  $b_1$ . In fact, any arbitrary mapping of events to channels is possible.

### 3.3 Algorithm for performance analysis

In this section, we present the performance analysis algorithm, which is shown in Figure 9). The algorithm *CommAnalyzer* traverses the entire *Communication Analysis Graph*, modifies it to account for the effects of the specified communication architecture, and assigns time-stamps to each node in the graph that reflect its execution time. *CommAnalyzer* results in modification of the CAG by

- Adding nodes, where necessary, to represent the hand-shaking required in order to initiate a communication (e.g., handshaking with an arbiter of a shared bus)
- Re-sizing a communication node into a sequence of nodes that respect the maximum block transfer size of the channel (e.g. splitting a large data transfer into DMA blocks)

During the traversal, *CommAnalyzer* maintains a set of lists, *ready\_nodes*, which contain the nodes that are ready to be executed (assigned time-stamps). The assignment of time-stamps accounts for dynamic delays due to bus contention/arbitration, in addition to considering the times required for the computation, communication, and handshaking nodes. The weight (execution time) of a computation node is derived from the initial co-simulation, while the weights of handshaking and communication nodes depend on the amount of data communicated, as well as the characteristics of the communication channel(s) it is mapped to.

The data structure *channel*, derived from a specification of the communication architecture, contains information about each channel. Channels are numbered 1 through `MAX_CHANNELS` and details of the channel  $b_i$  are stored in *channel[i]*. *Ready\_nodes* is a set of lists, one for each channel in the communication architecture. The first list, *ready\_nodes[0]* is a special list containing nodes that represent ready computation events while *ready\_nodes[i]*, ( $i > 0$ ) contains the set of nodes that represent ready communication events mapped to channel  $b_i$ . Nodes in each list are sorted by *start\_time* and priority is used as a secondary key. This ensures that if two components simultaneously raise a request for a shared channel, the one of greater priority is served first.

The algorithm proceeds by invoking *get\_channel* which examines the first node of every list and chooses to execute the one with the earliest *start\_time*, say  $v$ . If  $v$  is a *computation* node, i.e.,  $v$  was dequeued from *ready\_nodes[0]*, it is executed. The function *execute(v)* involves removing  $v$  from the list of *ready\_nodes* and marking it *visited*. Housekeeping operations involve adjusting the *start\_times* of its successors, and inserting any enabled nodes into an appropriate *ready\_nodes* list.

If  $v$  is a *communication* node, the node is scaled by *get\_weight* to incorporate the width and speed of the channel  $b_i$  to which it is mapped. On dequeuing the node from the  $i^{th}$  list in *ready\_nodes*, its actual size in terms of bus cycles is calculated from the width and speed of the channel  $b_i$ , and the original weight of the node in bytes.

Immediately after it is dequeued from *ready\_nodes[i]*, the *start\_time*  $t_1$  of a *communication* node  $v$  represents the time at which it makes a request for access to the channel  $b_i$ . If  $b_i$  is a shared bus, a check is made for the most recent node that accessed  $b_i$ . If it has a *finish\_time* of  $t_2$  (where  $t_2 > t_1$ ), the tool will delay the *start\_time* of the requesting node  $v$  to time  $t_2$ .

A *handshake* node is constructed to take into account the overhead of the protocol. The size of this node is dictated by the value of a constant *latency* whose value represents the intrinsic channel specific overhead preceding the actual communication. The *start\_time* of the *handshake* node is  $t_2$ , while the actual communication event is assigned a *start\_time* of  $t_2 + \textit{latency}$ . If  $b_i$  is a dedicated link, the check for concurrent access is not performed and the *handshake* node is not generated.

The size of the *communication* node, expressed in terms of bus cycles, is compared against the maximum permissible DMA size on the given channel. If the size is larger, a new node of size equal to the DMA size is created, inserted into the graph and executed. The *start\_time* and *weight* of the original node is modified and it is re-inserted into the same *ready\_nodes* list.

---

```

CommAnalyzer
inputs: CAG G, Communication Arch Description
outputs: CAG G, Performance Statistics,
Critical Path
initial
    mark all nodes visited = false;
    readArchitecture(channel[MAX_CHANNELS]);
    initialize(ready_nodes[MAX_CHANNELS], G);
begin
do
     $i := \textit{get\_channel}(\textit{ready\_nodes});$ 
     $v := \textit{deque}(\textit{ready\_nodes}[i]);$ 
    if  $v.type = \textit{COMP}$ 
        execute( $v$ );
        modify_successor_start_times( $v$ );
        add_enabled_nodes( $v$ );
    elseif  $v.type = \textit{COMM}$ 
        get_weight( $v.weight, \textit{channel}[i]$ );
        if  $\textit{channel}[i].type = \textit{shared}$ 
            if  $\textit{channel}[i].\textit{next\_free\_time} > v.start\_time$ 
                 $v.start\_time := \textit{channel}[i].\textit{next\_free\_time};$ 
            endif
            create( $w, \textit{HS}$ );
             $w.start\_time = v.start\_time;$ 
             $w.weight = \textit{channel}[i].\textit{latency};$ 
            insert( $w, G$ );
            execute( $w$ );
             $v.start\_time = w.finish\_time;$ 
        endif
        if  $v.weight < \textit{channel}[i].\textit{DMA\_SIZE}$ 
            execute( $v$ );
             $\textit{channel}[i].\textit{next\_free\_time} := v.finish\_time;$ 
            modify_successor_start_times( $v$ );
            add_enabled_nodes( $v$ );
        else
            create( $v', \textit{COMM}$ );
             $v.start\_time = v.start\_time;$ 
             $v'.weight = \textit{channel}[i].\textit{DMA\_SIZE};$ 
            insert( $v', G$ );
            execute( $v'$ );
             $v'.start\_time = v'.finish\_time;$ 
             $v.weight = v.weight - \textit{channel}[i].\textit{DMA\_SIZE};$ 
            insert( $v, \textit{ready\_nodes}$ );
        endif
    endif
until no more ready nodes
    generate_stats();
end

```

---

Figure 9: The CommAnalyzer algorithm.

Waiting for a free channel and resizing of nodes may change the *finish\_time* of the currently executing node  $v$ , and therefore the new *weight* plus the new *start\_time* is used to update the start-times of its successors. This is done by *modify\_successor\_start\_times*. Enabled successors (those successors all of whose predecessors have been executed) are then put into the appropriate list in *ready\_nodes* by *add\_enabled\_nodes*. This function also accumulates statistics that help determine the critical path.

The output of the algorithm includes execution statistics and performance and critical path information. The algorithm terminates

when all the nodes of the CAG are visited exactly once.

## 4 Experimental Results

In this section, we report on the application of the proposed analysis technique to evaluate alternative communication architectures of two example HW/SW systems.

Each example system is described as a set of communicating processes using Esterel and C in the POLIS [16] co-design framework. Subsequent HW/SW co-simulation of the system using PTOLEMY [17] assumes an abstract event-based modeling of communication. Traces are collected, converted into an equivalent CAG, and input to our analysis tool, along with a description of the communication architecture that is to be evaluated. In order to confirm the accuracy and efficiency of our analysis technique, the system performance is also measured by co-simulation of the entire system, including components and explicit hardware models of the communication architecture [18], using PTOLEMY.

The first set of experiments were conducted on SYS4.2, the four component system shown in Figure 2(a) and described in Section 2. Table 3 reports the results of performance analysis using both the proposed CAG-based analysis method, and the traditional approach of co-simulation on three alternative communication architectures, Case 1 - Case 3 shown in Figures 2(b)-(d).

The results in Table 3 demonstrate:

- The proposed performance analysis technique is accurate — the estimates obtained are within 3.5% of those obtained via co-simulation.
- Exploring various communication architectures can lead to significantly better design choices to improve the performance of a system. The results indicate that in Case 1, conflicts on the shared bus lead to poorer performance (about 26%) than Case 3, where use of two buses exploits the synchronization between the components to minimize conflicts on each of shared buses. However even though Case 2 uses multiple buses, the additional cost of communicating through a bridge by for each memory request by  $C_2$  and  $C_3$  degrades performance by 9%.

Our second example system, shown in Figure 10(a), is a three component system that is part of a TCP/IP network interface card. It performs a checksum computation on packets arriving from the network. Each packet is written to memory by CREATE\_PACKET, which then signals IP\_CHK to overwrite a part of the header with zeros (the part that should not be used in the checksum computation). Once IP\_CHK is done, it passes CHKSUM the old value of the checksum and signals it to calculate the new value by processing the bits stored in memory. If the two do not match an error is raised.

Two alternative communication architectures are shown: Figure 10(b), using a shared bus (Case 1), and Figure 10(c), using three

Table 3: Comparison of alternative architectures of SYS4.2

SYS4.2 Communication Architecture	CAG Analysis Estimate (cycles)	Co-simulation Estimate (cycles)	Error %
Config = Case 1, Bus: width = 8, DMA = 50, C1 > C2 > C3 > C4, Latency = 1	224031	231970	-3.42
Config = Case 2, Bus <sub>1</sub> : width = 8, DMA = 50, C1 > C2, Latency = 1, Bus <sub>2</sub> : width = 8, DMA = 50, C3 > C4, Latency = 1	244002	245696	-0.70
Config = Case 3, Bus <sub>1</sub> : width = 8, DMA = 50, C1 > C3, Latency = 1, Bus <sub>2</sub> : width = 8, DMA = 50, C2 > C4, Latency = 1	165987	169999	-2.36

buses (Case 2). Table 4 shows results of simulating the system under one configuration of Case 1 and for two configurations of Case 2.

Table 4: Comparison of alternative architectures of TCP/IP

TCP_IP Communication Architecture	CAG Analysis Estimate (cycles)	Co-simulation Estimate (cycles)	Error %
Config = Case 1, Bus width = 32, DMA_SIZE = inf., CREATE_PACKET >> IP_CHK >> CHKSUM, Latency = 1	69013	69508	-0.71
Config = Case 2, bus <sub>1</sub> = bus <sub>2</sub> = bus <sub>3</sub> : width = 32, DMA_SIZE = inf., CREATE_PACKET >> IP_CHK >> CHKSUM, Latency = 1	35079	36075	-2.76
Config = Case 2, bus <sub>1</sub> = bus <sub>2</sub> = bus <sub>3</sub> : width = 16, DMA_SIZE = inf., CREATE_PACKET >> IP_CHK >> CHKSUM, Latency = 1	67645	66636	1.51

Again, it is clear that no loss of accuracy has been suffered. Additionally, Table 4 shows a comparison of the performance of the system under three situations. The shared bus of 32 bits provides poorer performance (49.2%) than a multiple bus configuration of 3 buses that allow uninterrupted pipelined processing of incoming packets. (While packet  $i$  is accessed by CHKSUM in MEM3 via Bus 3, packet  $i + 1$  is accessed by IP\_CHK in MEM2 via Bus 2, and packet  $i + 2$  is written to MEM1 via Bus 1). However when the width of each of the three buses was reduced to 16, the performance improvement dropped to only 2%, showing that the advantage of splitting the bus and thereby decreasing conflicts is heavily countered by the price of lower bandwidth on each bus.

Table 5 compares the efficiency of our technique versus complete system co-simulation for 4 of the configurations studied. The first column denotes the time taken by the first phase (co-simulation which uses an abstract event-based model of communication). The second column shows the time taken by our analysis tool to generate performance figures for a given CAG and communication architecture. The third column denotes the time taken for the entire system to be co-simulated, including behavioral HW models of the communication architecture. Also note that system co-simulation using the hardware models of the communication architecture (third column of Table 5) takes significantly more time than co-simulation performed with an abstract model of communication (second column of Table 5).

Measurements of elapsed time were made on a single user Sun Ultra-II workstation running Solaris 2.6. In row 3 it is observed that the speed up of using our analysis over conventional HW/SW co-simulation is 245. In general, Table 5 reports a speed up of 2 orders of magnitude for each system investigated. Note that the first column entries of rows 2 and 3 of Table 5 are zero because only one initial co-simulation (that reported in row 1) was necessary in order to explore all the alternative communication architectures. Hence, evaluating several alternative communication architectures using our analysis technique is accurate and fast, while it would be prohibitively time-consuming using the system co-simulation approach (fourth column).

## 5 Conclusions

In this paper we have demonstrated that the selection of communication architectures — their topology and per channel characteristics — can have a significant impact on the performance of a system, motivating the need for fast and accurate exploration tools to evaluate various design alternatives. We have described a trace-based approach that we believe will be helpful in meeting this need in the SOC design environment, and have presented experimental results that support claims of efficiency and accuracy. In future work we intend to address the issue of how to incorporate the

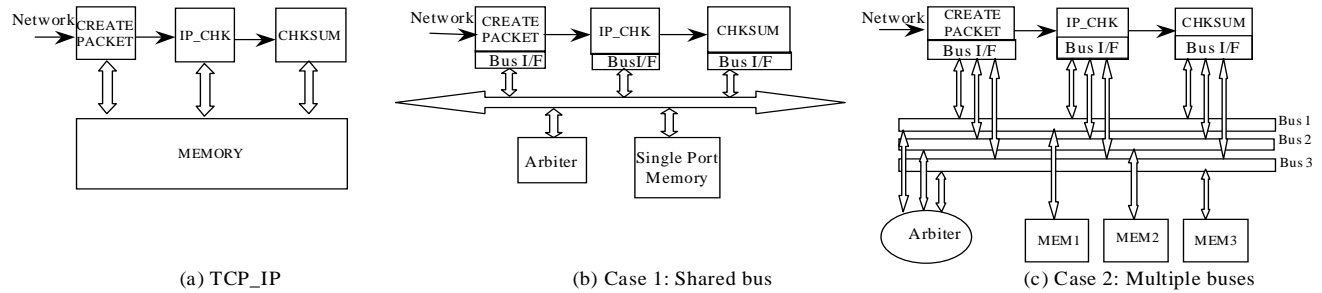


Figure 10: Alternative implementations of TCP\_IP

Table 5: Comparison of running time of CAG based analysis against co-simulation

Case Study	Co-simulation for CAG Generation (seconds)	CAG Based Analysis (seconds)	System co-simulation (seconds)	Speedup
SYS4.2 in Case1 configuration	220	8.2	1325	162
SYS4.2 in Case2 configuration	0	8.3	1895	228
SYS4.2 in Case 3 configuration	0	7.6	1863	245
TCP_IP in Case 1 configuration	84	3.0	688	229

proposed analysis tool into a optimization framework that will automatically generate an optimal communication architecture for a given system.

## References

- [1] D. D. Gajski, F. Vahid, S. Narayan and J. Gong, *Specification and Design of Embedded Systems*. Prentice Hall, 1994.
- [2] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design & Test Magazine*, pp. 64–75, Dec. 1993.
- [3] T. B. Ismail, M. Abid, and M. Jerraya, "COSMOS:A codesign approach for a communicating system," in *Proc. IEEE International Workshop on Hardware/Software Codesign*, pp. 17–24, 1994.
- [4] A. Kalavade and E. Lee, "A globally critical/locally phase driven algorithm for the constrained hardware software partitioning problem," in *Proc. IEEE International Workshop on Hardware/Software Codesign*, pp. 42–48, 1994.
- [5] P. H. Chou, R. B. Ortega, and G. B. Borriello, "The CHINOOK hardware/software cosynthesis system," in *Proc. Int. Symp. System Level Synthesis*, pp. 22–27, 1995.
- [6] B. Lin, "A system design methodology for software/hardware codevelopment of telecommunication network applications," in *Proc. Design Automation Conf.*, pp. 672–677, 1996.
- [7] J. A. Rowson and A. Sangiovanni-Vincentelli, "Interface Based Design," in *Proc. Design Automation Conf.*, pp. 178–183, June 1997.
- [8] K. Hines and G. Borriello, "Optimizing Communication in embedded system cosimulation," in *Proc. International Workshop on Hardware/Software Codesign (codes/CASHE)*, pp. 121–125, Mar. 1997.
- [9] T. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, pp. 288–294, Nov. 1995.
- [10] J. Daveau, T. B. Ismail, and A. A. Jerraya, "Synthesis of system-level communication by an allocation based approach," in *Proc. Int. Symp. System Level Synthesis*, pp. 150–155, Sept. 1995.
- [11] S. Dey and S. Bommu, "Performance Analysis of a system of communication processes," in *Proc. Int. Conf. Computer-Aided Design*, pp. 590–597, Nov. 1997.
- [12] P. Knudsen and J. Madsen, "Integrating communication protocol selection with partitioning in hardware/software codesign," in *Proc. Int. Symp. System Level Synthesis*, pp. 111–116, Dec. 1998.
- [13] M. Gasteier and M. Glesner, "Bus-based communication synthesis on system level," in *ACM Trans. Design Automation Electronic Systems*, pp. 1–11, Jan. 1999.
- [14] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, San Mateo, CA, 1989.
- [15] "WARTS: Wisconsin Architectural Research Tools Set, Computer Science Department, University of Wisconsin. (<http://www.cs.wisc.edu/larus/warts.html>)."
- [16] F. Balarin, M. Chiodo, H. Hsieh, A. Jureska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki and B. Tabbara, *Hardware-software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, Norwell, MA, 1997.
- [17] J. Buck and S. Ha and E. A. Lee and D. D. Masserchmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal on Computer Simulation, Special Issue on Simulation Software Management*, Jan. 1990.
- [18] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, and A. Sangiovanni-Vincentelli, "A Case Study on Modeling Shared Memory Access Effects During Performance Analysis of HW/SW Systems," in *Proc. International Workshop on Hardware/Software Codesign (codes/CASHE)*, Mar. 1998.