To appear in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), November 1999

Fast Performance Analysis of Bus-Based System-On-Chip Communication Architectures [*]

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

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

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

Abstract

This paper addresses the problem of efficient and accurate performance analysis to drive the exploration and design of bus-based System-on-Chip (SOC) communication architectures. 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). The proposed system-level performance analysis technique consists of (i) initial co-simulation performed after HW/SW partitioning and mapping, with the communication between components modeled in an abstract manner (*e.g.*, as events or data transfers), (ii) extraction of abstracted symbolic traces, represented as a Bus and Synchronization Event (BSE) Graph, that captures the activity of the various system components and their communication over time, and (iii) manipulation of the BSE Graph using the bus parameters, to derive the behavior of the system accounting for effects of the bus architecture. We present experimental results on several example systems, including a TCP/IP network interface card sub-system. The results indicate that our performance estimation technique is over two orders of magnitude faster than performing a complete system simulation, while being very accurate (within 2.2% of performance estimates derived from accurate HW/SW co-simulation).

I. Introduction

Realizing the complete potential of SOC design depends heavily on the availability of design tools and methodologies that help the system designers to explore system-level tradeoffs. Specifically, the availability of fast and accurate analysis and modeling techniques for metrics such as performance, power, and cost, is critical to guide various design decisions. In this paper, we focus on performance analysis to support the design of bus-based SOC communication architectures.

While a large body of research on system synthesis has focussed on scheduling, partitioning, and mapping the target application functionality to an optimal set of system components, often equally important is the choice of communication architectures for the SOC. The SOC communication architecture determines the way in which the components communicate with each other to synchronize and exchange data. For example, the choice of the bus architecture and protocol to be used, the system memory organization to be used, *etc.*, can also be selected and customized for the given application. Various studies (*e.g.* [1]) have demonstrated the impact of the SOC communication architecture on the system's performance. The benefits of separating the interfaces of components from their internal behavior/computation, starting with the system specification stage through the system refinement stages, were presented in [2].

Various techniques have been proposed for performance analysis of hardware [3, 4] and software [5, 6]. System-level performance analysis techniques which consider the effects of communication 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 [2, 7]. 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 [4, 8, 9, 10, 11]. 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.* [10, 11]), or are overly pessimistic by assuming a worst-case scenario for bus contention (*e.g.* [8]).

A. Problem Overview and Paper Contributions

In this subsection, we first highlight some important features of bus-based communication architectures, and their implications on system-level performance analysis. Later, we explain how these issues are addressed in the techniques proposed in this paper.

Bus-based architectures are very commonly used to facilitate communication between the various system components [12]. Since buses are shared communication channels, they require arbitration (through a *bus arbiter*) in order to ensure that only one component has control of the bus at a time. Thus, a component that wishes to transfer data over the bus needs to first *handshake* with the arbiter. When multiple components seek to use the bus simultaneously, the arbiter decides (typically based on a *priority scheme*) which component is granted the right to access the bus. In order to facilitate efficient transmission of larger chunks/bursts of data, buses may also provide a *DMA* or *block transfer mode* wherein a component pre-negotiates the right to use the bus for multiple bus cycles. In order to prevent any one component from monopolizing the bus and introducing a high latency to other component's access requests, a

[*]This work was supported by NEC USA Inc., and by the California Micro Program

maximum DMA block size is typically placed on the amount of data that can be transmitted as a single DMA block.

The above factors make the estimation of the system performance in the presence of the bus a complex task. The time taken by a component to transmit a piece of data over the bus depends not only on the amount of data, but also on the bus protocol and values of bus parameters mentioned above, and on the activity (bus access requests) from other system components. As a further complication (as shown later in Section III), not only does the bus architecture directly impact the time taken for a communication across the bus, but it also indirectly impacts the timing of the other operations performed by the system components. In order to account for these indirect effects, it is imperative to analyze the bus architecture in conjunction with the other parts of the system.

The *contribution of this paper* is an efficient and accurate system performance estimation technique for evaluation of SOC bus architectures. Our technique is based on a two-phase approach. An initial co-simulation is performed using information about the refinement of the system components (HW/SW partitioning and mapping), but with the communication between components modeled in an abstract manner. From this initial phase, we extract abstracted symbolic traces (represented as a *Bus and Synchronization Event Graph*) that capture the activity of the various system components and their communication over time. In the second phase, these traces are manipulated using the values of the various bus parameters selected by the designer (*e.g.*, bus width, priorities, support for DMA, DMA block size, arbiter handshake overhead, *etc.*), to derive the timing behavior of the system accounting for the effects of the bus architecture.

We have implemented our performance analysis tool and validated it on several example systems, including a TCP/IP network interface card sub-system. The results indicate that our performance estimation technique is *over two orders of magnitude faster* than performing a complete system simulation, while being *very accurate* (within 2.2% of performance estimates derived from accurate HW/SW co-simulation).

The rest of the paper is organized as follows. Section II presents further motivation for our performance estimation technique, Section III presents an overview of our performance analysis technique, and how it fits into a generic system design methodology/flow. Section IV presents some algorithmic details of our performance estimator. Section V presents experimental results and conclusions.

II. Effect of Bus Architectures and Protocols on System Performance

In this section, we present effects of customizing bus-based communication architectures on the performance of a hardware/software system-on-chip through several experiments. Our investigation motivates the need for fast and accurate performance analysis to enable efficient exploration of the system design space, and selection of bus architectures and protocols to optimize system performance.

We start by analyzing the performance of a TCP/IP Network Interface System [1] under differing memory and communication architectures. The sub-system consists of the part of the TCP/IP protocol related to the checksum computation (Figure 1). For incoming packets, the task *Create_Packet* receives a packet and stores it in a memory. When it finishes, it sends the information about the starting address of the packet in memory, the number of bytes and the checksum header to the *Packet_Queue*. From this queue, *IP_Chk* retrieves a new packet, overwrites parts of the checksum header (which should not be used in the checksum computation) with 0s, and signals to the *Chksum* task that a new packet

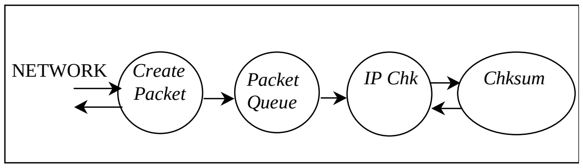

Figure 1: The TCP/IP Network Interface System

can be checked for checksum consistency. *Chksum* performs the core of the computation, accessing the packet in memory and accumulating the checksum for the packet body. When it is done, it sends the computed 16-bit checksum back to the *IP_Chk* task, which then compares the computed checksum with the incoming transmitted checksum, and flags an error if they do not match. The flow for outgoing packets is similar, but in the reverse direction, and there is no need for comparison of the final checksum.

A. Effect of Bus Architectures

Figure 2 shows one partitioning and mapping, where the tasks *Create_Packet* and *Packet_Queue* are software tasks and are mapped to a MIPS R3000 processor, while the remaining tasks *IP_Chk* and *Chksum* — are mapped to dedicated hardware. While most previous research has concentrated on HW/SW partitioning and mapping, we show that choosing an optimal communication architecture is also critical to the system performance. Figure 2(a) shows a candidate communication architecture where the system components access a shared multi-port memory through dedicated links.

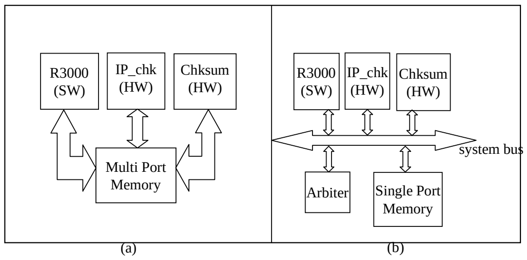

Figure 2: Alternative communication architectures

This architecture allows the packets arriving serially into the system to be processed in a pipelined fashion by the tasks. While *Chksum* is processing packet i, *IP_Chk* can process packet $i+1$ and *Create_Packet* can be working on packet $i+2$ all at the same time. At any given moment, the various tasks in the system access different parts of the memory because each is operating on a different packet. Hence, the concurrent tasks can operate without any conflict, resulting in superior performance.

An alternative architecture is shown in Figure 2(b). Here the components of the system access a shared single-port memory through a common system bus. In this shared bus architecture, an arbiter resolves conflicts resulting from simultaneous attempts to access the bus.

We have performed experiments to observe the performance of the TCP/IP System under various memory and communication architectures using POLIS [13] as the Hardware/Software co- design tool, and PTOLEMY [14] for system-level simulation. We have used a behavioral bus arbiter model [1] to take into account the effect of the bus architecture on system performance.

Our experimental results show that the *processing time per packet of each component for the shared bus architecture of Figure 2(b) is upto* 40% *higher than that for the dedicated link case of Figure 2(a)*. The degradation is because the shared bus introduces *"conflict waiting time"* whenever two components simultaneously request access to memory, whereas in a dedicated bus architecture the components are permitted to concurrently access memory. These results demonstrate the significant impact of bus and memory architectures on system performance. They also show the importance of making a judicious selection of the communication architecture when mapping an application to produce a high performance application-specific system-on-chip.

B. Effect of Bus Protocols on Performance

Our next experiments show that even after the bus and memory architectures have been selected and fixed, the bus protocols and parameters used can greatly influence the performance of the system. Consider the execution traces of two packets by the TCP/IP tasks, *Create_Packet*, *IP_Chk*, and *Chksum*, under three different cases, as shown in Figure 3. Case 1 reflects the case when the second packet can be processed by *Create_Packet* and *Chksum*, without any conflict. This execution trace can be generated by using a dedicated link architecture like Figure 2(a). Cases 2 and 3 are possible execution traces under a shared bus architecture, the difference being in the way the priorities have been assigned. Assuming static priority based arbitration, for Case 2 *Create_Pack* is assigned the highest priority among the competing tasks while in Case 3 it has the lowest. Figure 3 shows that the times at which processing of each packet is completed by the different tasks depends not only on the bus architecture used (dedicated *vs.* shared), but also the task priorities used. Depending upon the composition of the system critical path from the different tasks, the system performance can be greatly affected [4]. In general, besides task priorities, other bus parameters like the DMA size used for bus/memory transfers, significantly affect the system performance, as will be illustrated by the next experiment.

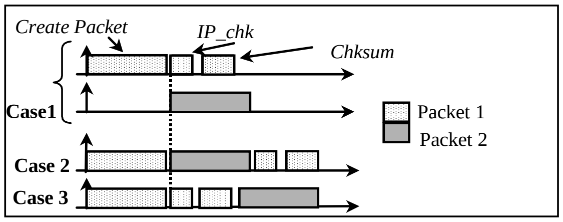

Figure 3: Variation of processing time with priority assignments

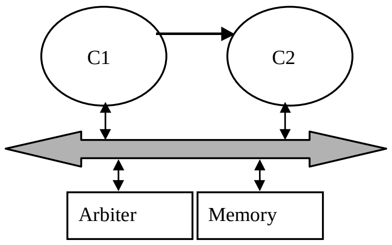

Figure 4: Example system to illustrate effect of DMA size on performance

Effect of DMA size on system performance: Consider a simple system shown in Figure 4 that consists of two components, $C1$ and $C2$, 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 in a manner similar to that described for the TCP/IP sub-system. The system supports DMA mode transfers across the shared bus.

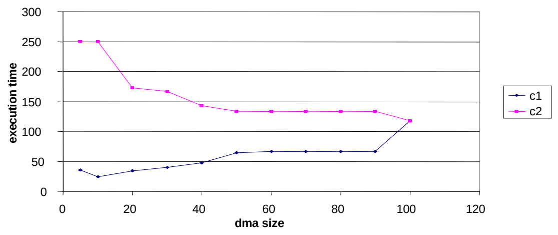

Figure 5: Effect of DMA size on performance

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 $C1$ performs computations of average size 10 cycles and memory transfers of average size 10 bus words while $C2$ performs computations of average size 10 but memory transfers of average size 100. Figure 5 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 5 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 $C1$.

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.

III. Overall Performance Estimation Methodology

Our tool for evaluating SOC bus-based communication architectures is based on a two-phase performance estimation methodology for HW/SW system design. In a generic system design flow, we envision that performance estimation will be used to (i) support the refinement (*e.g.*, HW/SW partitioning and mapping) of various parts of the system functionality, and (ii) refinement of the communication between the system components. In the first phase, conventional system-level performance analysis tools such as HW/SW co- simulation could be used, with the communication between components described and simulated in an abstract manner (*e.g.*, as abstract data transfers or events). The performance analysis technique presented in this paper fits into the second phase, thus it complements most of the existing technology for system-level performance analysis. Please note that while we treat these phases to be separate in this paper, it is conceivable to consider them in an integrated manner as a single system-level performance analysis tool.

An overview of our tool's inputs and outputs, and of how we have integrated it into an existing HW/SW system design flow, are provided in this section. We have currently integrated our performance estimator into the POLIS [13] and PTOLEMY [14] co-design environment, although our techniques can also be used in conjunction with other co-design frameworks as well. The system specification (a set of communicating processes) is partitioned (manually or automatically) into HW and SW, and possibly parts

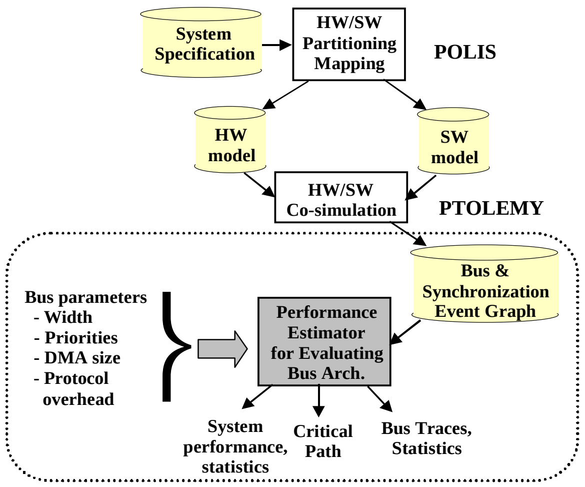
System Specification
HW/SW Partitioning Mapping
POLIS
HW model
SW model
HW/SW Co-simulation
PTOLEMY
Bus & Synchronization Event Graph
Bus parameters - Width - Priorities - DMA size - Protocol overhead
Performance Estimator for Evaluating Bus Arch.
System performance, statistics
Critical Path
Bus Traces, Statistics

Figure 6: Overview of proposed performance analysis methodology

of it are mapped to pre-designed cores. For example, a processor core is selected to implement the SW parts. As is the case in several current co-design systems, the above steps are performed without specifying anything about the system communication architecture. HW/SW co-simulation, using the PTOLEMY simulation environment, provides performance estimation. The co-simulation uses an (event-based) abstract model of communication between system components [13].

Our tool extracts, from the HW/SW co-simulation, a set of *symbolic computation and communication traces* for each component in the partitioned and mapped system specification. We represent these traces using a data structure called the *Bus and Synchronization Event Graph* (BSE graph). The BSE graph is a vertex-weighted directed acyclic graph with the following properties:

- Vertices represent computations, data transfers over the bus, and synchronization between components.
- The weight of a vertex is the duration of the computation, data transfer, or synchronization.
- Edges represent precedence constraints between vertices. An edge exists from $v1$ to $v2$ if the computation or communication represented by $v2$ cannot occur until that represented by $v1$ has completed. Edges arise due to the sequentiality of operations within a component or due to inter-component communication [4].

In the BSE graph generated from the initial HW/SW co- simulation, each abstract data transfer over the bus is represented as a single vertex. Since communication is modeled as exchange of abstract events during the initial co-simulation, no information about the timing of communication vertices is available in the initial BSE graph.

The performance estimator takes as inputs the BSE graph, and values of various bus parameters chosen by the designer. It generates an augmented BSE graph that captures the behavior of the system incorporating the bus effects. The augmented BSE graph contains additional vertices (introduced by our tool) that represent protocol overhead and synchronization overhead. In addition, the estimator ensures that at most one data transfer is active for the bus at any given time (conflict resolution/arbitration), and splits bus transfer vertices into multiple smaller vertices, if necessary, due to the DMA block size restriction. While traversing and manipulating the BSE graph, the estimator computes a *time-stamp* for each vertex in the augmented BSE graph. The time-stamps of the vertices are used to generate various outputs including:

- System performance statistics incorporating the effects of the bus architecture. Performance statistics can include completion times of specific computation(s), generation times of specific event(s), separation between specified event(s), *etc.*
- The system critical path, which can run through multiple components [4].
- A symbolic system execution trace that indicates the activity of the system over time (the trace is symbolic in the sense that the actual data values are not included since they are abstracted out in the generation of the BSE graph).
- Bus-related statistics such as the amount of time each component spends waiting for the bus, handshaking with the arbiter, and waiting for synchronization events from other components.

The designer can use the results of performance estimation to modify the bus architecture and thus iteratively explore the design space for system-level bus-based communication architectures.

A. Accuracy and Efficiency Issues

The efficiency of our performance estimation tool is derived from the fact that we abstract out the details of the computations and communications (bus accesses) between the system components, and cluster them into vertices while deriving the BSE graph. For example, in the case of a computation vertex, we only care about the difference between its start and finish times. In the case of a bus access vertex, we only care about the amount of data transferred. As a result, a BSE graph that represents millions of cycles of execution of an entire system might contain only hundreds of vertices and edges. This abstraction is especially necessary since the BSE graph is constructed from the dynamic traces resulting from co-simulation, *i.e.*, it represents the execution of the system *“unrolled in time”*. Overall, the computational complexity of our performance estimator is *linear in the size of the BSE graph*, hence much faster than complete system co-simulation. The efficiency of our performance analysis technique is further borne out by the experimental results presented in Section V.

The accuracy of our performance estimation technique is due to two factors:

- Since we are using a dynamic execution trace derived from co-simulation, the control flow within each component is fully determined (*e.g.*, we don't need to worry about predicting how many times each loop is executed, how each branch is taken *etc.*).
- Since we are not isolating the bus accesses from the rest of the system (computation vertices, synchronizations), it is possible to account for the direct effects as well as indirect effects of the bus architecture. The importance of accounting for such indirect effects is illustrated later in this section.

It bears mentioning at this point that we assume that the actual operations performed in the computation nodes and the data transferred in the memory access nodes are not dependent on the bus effects. Put in a different way, the bus architecture can affect the timing of the various computations and communications in the system, but not the functionality. We believe that this assumption is quite general, and is similar to assumptions made in several typical system design methodologies/tools [13, 15].

The following example illustrates the importance of considering indirect effects of the bus architecture on the timing of the various system components. In particular, we focus on the effects

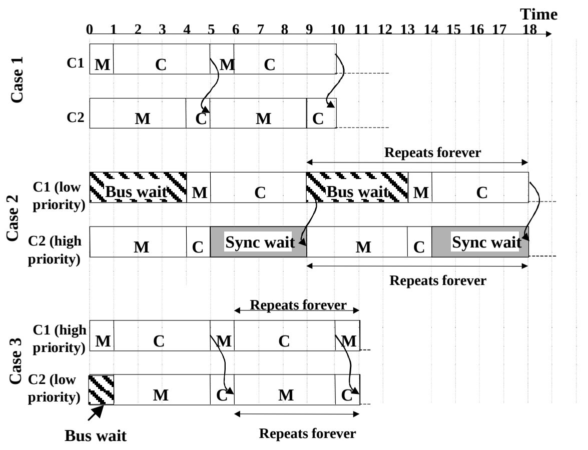

Figure 7: Indirect effect of bus architecture on synchronization wait time

of bus wait time on the synchronization wait times of the various components.

Example 1: Consider again the two component system shown in Figure 4. Recall that components $C1$ and $C2$ access a global memory though a shared bus and synchronize with each other. The traces shown in Figure 7 represent the operation of the system of Figure 4 under three different scenarios. The first set of waveforms (Case 1) represent the activity of $C1$ and $C2$, as derived from a simulation of the system where the data transfers between the components and the memory are modeled in an abstract manner (as events). Thus, the first set of waveforms do not account for effects of the bus architecture. The arcs between the waveforms indicate the synchronization dependencies between the components. The second set of waveforms (Case 2) are derived from a simulation of the system with a model of the shared bus and bus arbiter, where $C2$ was assigned higher priority for bus access. Due to bus access conflicts, component $C1$ has to wait for accessing the bus when $C2$ (the higher priority component) also requests bus access. Thus, *bus wait* times are introduced in the waveform for component $C1$ from time unit 0 to 4, 9 to 13, and so on. An indirect effect of the bus wait times of component $C1$ is to introduce *synchronization wait* times for component $C2$ from time unit 5 to 9, 14 to 18, *etc*. Finally, the third set of waveforms (Case 3) were derived by assigning $C1$ higher priority for bus access. Since the bus parameters have changed, the bus wait times are now different (time unit 0 to 1 for component $C2$). However, in addition, note that the inter-component synchronization wait time has also been eliminated.

The above example indicates that merely considering the direct effects of the bus architecture without considering the indirect effects on the timing of the system could lead to erroneous performance estimates.

IV. Implementation of the Analysis Technique

Our performance estimation algorithm is described be the procedure in Figure 8. $Performance\_analyzer$ takes two inputs: G, the BSE graph, and *PARAMS*, a data structure containing the bus parameters. G is a vertex-weighted directed acyclic graph containing vertices of two types: *BUS* and *COMP*, representing bus accesses and computations respectively. During the execution of $Performance\_analyzer$ new vertices of two further types, *HANDSHAKE* and *SYNCH*, are introduced. Each vertex in G contains the following information: the type of vertex, its weight, time-stamp (start-time), list of predecessors, list of successors, and a flag *done* to denote whether or not the vertex has been executed. *PARAMS* contains the following integer parameters:

- BUS_WIDTH in bytes
- SYNCH_OVERHEAD — elapsed cycles between emission and consumption of a synchronization event
- PROTOCOL_OVERHEAD — elapsed cycles between the sending of a request to the arbiter and the receipt of a grant in the absence of conflicts
- MAX_DMA_SIZE — the largest number of bus words that a component is allowed to transfer per grant. It is assumed (without loss of generality) that exactly one bus word can be transferred per clock cycle.
- PRIORITIES — an array of integers specifying the static priority of each component in the system.

```
Performance_Analyzer(G, PARAMS)
begin
Ready_vertex_list := Create_ready_vertex_list(G);
while (Ready_vertex_list is not empty) do
   v := Dequeue(Ready_vertex_list);
   if (v,u) is a synchronizing edge, create w of type SYNCH;
       w.weight := PARAMS.SYNCH_OVERHEAD;
       Insert_edge(v,w);Insert_edge(w,u);Delete_edge(v,u);
Case1: v is of type COMP or SYNCH
         mark v.done = TRUE; Modify_successors (v);
         Update_ready_vertex_list(v);
Case 2:v is of type BUS
         v.weight = v.weight / PARAMS.BUS_WIDTH;
         Delay_bus_access(v);
         Create  a new vertex w of type HANDSHAKE;
         w.weight := PARAMS.PROTOCOL_OVERHEAD;
         For all  u in v.predecessors, Insert (u,w), Delete (u,v);
         Insert (w,v);
         Case 1:v.weight<= PARAMS.MAX_DMA_SIZE
                  mark v.done := TRUE;
                  v.timestamp := w.timestamp+
                  PARAMS.PROTOCOL_OVERHEAD;
                  Modify_successors(v); last_bus_access := v;
                  Update_ready_vertex_list(v);
         Case 2: v.weight > PARAMS.MAX_DMA_SIZE
                  Create w, w.weight := v.weight – PARAMS.
 MAX_DMA_SIZE; v.weight:= PARAMS.MAX_DMA_SIZE;
                 For all u in  v.successors, insert (w,u), delete (v,u);
                 Insert(v,w);last_bus_access := v; Enqueue (w);
end while
end
```

Figure 8: Performance analysis algorithm

The procedure $Performance\_analyzer$ behaves as follows. All vertices are initially marked with $done = FALSE$. This is done during construction of the BSE graph. The $Ready\_vertex\_list$ is a list of vertices ordered on two keys — the primary key is the time-stamp and the secondary key is the priority of the component. It may be noted that the size of this list never exceeds the total number of components in the system because each component can have only one ready vertex at any given time. An initial $Ready\_vertex\_list$ is constructed by the function $Create\_ready\_vertex\_list(G)$ from vertices in the input graph G that have no predecessors.

From here on $Performance\_analyzer$ keeps dequeuing vertices from the $Ready\_vertex\_list$ till it there are no more ready vertices. This will happen only when the entire graph has been traversed exactly once. Each vertex that is dequeued is examined:

if it has a synchronizing edge emanating from it, a *SYNCH* vertex is placed on the edge to represent the overhead of synchronization. If the vertex is of type *COMP*, it is executed, the time-stamps of its successors are modified using *Modify_successors*, and finally *Update_ready_vertex_list* is called to add enabled vertices to the list of ready vertices.

If the vertex is of type *BUS*, the weight of the vertex (which represents the size of the transfer in bus words) is scaled in inverse proportion to the bus width. A *HANDSHAKE* vertex is generated of appropriate weight and if necessary (depending on the size of the bus vertex) the original *BUS* vertex is split it into two *BUS* vertices: one of weight *MAX_DMA_SIZE* and the other of weight equal to the fragment not served. The first vertex is executed while the second goes back into the *Ready_vertex_list*. In addition, for *BUS* vertices, we need to examine *last_bus_access*, which records the last vertex that was granted access to the bus. The time-stamp of a *BUS* vertex is determined from the finish time of the *last_bus_access* by the procedure *Delay_bus_access* which captures the implicit precedence that exists between competing bus access vertices.

Delay_bus_access and *Modify_successors* also accumulate statistical information regarding the critical vertices, wait time due to synchronization and wait time due to bus conflicts.

For a specified set of parameters the following information can be obtained using the data accumulated during the traversal of the BSE graph:

- The performance of the system for the chosen set of parameters. The time-stamp on the last node to be inserted into the augmented BSE graph gives us a performance measure. In addition we can examine time-stamp values at which specific computations complete or specific events are generated. That is, we can regenerate an abstract system execution trace.
- The system critical path is easily obtained. The functions *Modify_successors* and *Delay_bus_access* record critical predecessors for each vertex. After the algorithm completes, we back up and traverse the critical predecessor links starting from the last executed vertex to get the system critical path.
- The percentage of the total execution time each component spends waiting for:
  - Synchronization with other components — This information is accumulated in an array *synch_wait*. $synch\_wait[p,q]$ is updated in the *Modify_successors* routine. Let the critical predecessor of an vertex v in component p be a *SYNCH* vertex w sent by component q. The difference between the finish times of w and v's second most critical predecessor gives the time v spends waiting for the synchronizing event w. This is added to the time p has so far spent waiting for q by incrementing $synch\_wait[p,q]$.
  - Protocol overhead — This is obtained by counting the number of handshake type vertices introduced for each component and multiplying by the constant *PROTOCOL_OVERHEAD*.
  - Bus access conflicts — When the critical predecessor of a vertex v of a component p is *last_bus_access*, p has waited on a bus conflict. The duration of this wait is the difference between the finish times of *last_bus_access* and the v's second most critical predecessor. Such waits are accumulated for component p in $bus\_wait[p]$ by the function *Delay_bus_access*.

The running time of *Performance_analyzer* is linear in the size (no of vertices and edges) of the BSE graph. The arbitration and housekeeping operations require $O(M)$ time for every vertex that is scheduled, where M is the maximum number of components. However in reality the number of edges emanating from a vertex will be far less than M. We demonstrate both accuracy and efficiency of our technique in the next section by comparing our results with a fully specified system co-simulation.

V. Experimental Results

In this section we first demonstrate the accuracy and efficiency of our technique by comparing it against detailed HW/SW co-simulation for some example systems. We then show how our technique can be used for fast design space exploration when choosing bus parameters to maximize performance.

We used three example systems in our experiments — the TCP/IP network interface card sub-system of Figure 1 and two systems that are similar to the two component system of Figure 4. The latter two systems differ from each other significantly in their computation and bus access profiles. All systems were specified using Esterel and C and graphical schematic capture was performed in the POLIS/PTOLEMY framework. For each system an arbitrary set of values were chosen for the parameters of the bus architectures. For the TCP/IP system we simulated 100 packets, each of size 512 bytes with the bus parameters chosen as follows: *MAX_DMA_SIZE* = 16 bus words, *PROTOCOL_OVERHEAD* = 1 cycle, *BUS_WIDTH* = 8 bytes, and *SYNCH_OVERHEAD* = 1 cycle. The priorities were set so that *Create_Packet* has the highest priority followed by *IP_Chk* followed by *Chksum*. In the second system (*MEM_SYS*1) components $C1$ and $C2$ each have a mean bus access of size 10 and a mean computation of size 10. DMA block size was set to 5, the priorities were arbitrarily assigned, and the system was studied for an execution trace containing 2000 accesses from each component. The remaining parameters were set to the same value as in the TCP/IP system. In the third system, (*MEM_SYS*2) $C1$ has an average bus access of width 100 while the other has an average bus access width of 10. The system is studied for 2000 iterations of the $C1$ running concurrently with 400 iterations of $C2$.

Table 1: Accuracy of the proposed estimation technique

Example System	Co-simulation estimate (cycles)	BSE graph estimate (cycles)	% variation
TCP/IP	22877	22997	0.05
MEM_SYS1	68146	67827	0.47
MEM_SYS2	69858	71400	2.21

Table 2: Efficiency of the proposed estimation technique

Example System	Co-simulation Elapsed time (sec)	BSE graph Elapsed time(sec)
TCP/IP	87	0.05
MEM_SYS1	922	0.22
MEM_SYS2	638	0.13

For each system we performed two experiments to evaluate the system performance while incorporating the effects of the bus architecture. In the first experiment, we used a complete system co-simulation, with a behavioral model of the bus architecture [1]. These results appear in column 1 of Table 1 and Table 2. In the second experiment, we used the proposed performance analysis technique (Section III and Section IV) to estimate the total system performance. Table 1 and Table 2 present the results of our experiments. Table 1 reports the performance estimates obtained by complete system co-simulation (second column), the performance estimate obtained using our analysis technique (third column) and the percentage difference between the two (fourth column). Table 2 reports the efficiency (execution time) of a complete system co- simulation (second column), as well as our performance analysis technique (third column).

The results of Table 1 indicate that our technique has a negligible loss of accuracy compared to complete HW/SW co- simulation. We note that the difference in the estimated performance is no more than 2.21% for the cases studied. In the case of the TCP/IP study, there is only a 0.05% difference in the performance estimate of our tool versus that obtained from a complete system simulation using the POLIS/PTOLEMY framework.

Table 2 shows that our performance analysis technique is two to three orders of magnitude faster than complete HW/SW co-simulation. It is not inconceivable that for more complex systems than the ones we have studied, the speed-up will be even more significant due to the greater advantage of abstraction.

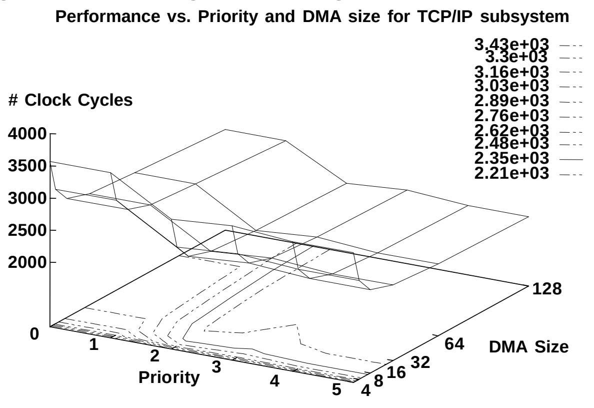

Figure 9: Efficient design space exploration for the TCP/IP system using the proposed estimation technique

In order to demonstrate the utility of our performance analysis technique in an iterative design space exploration framework we performed the following experiment. We ran an exhaustive search of all possible values of priority assignments and all meaningful DMA block sizes for the TCP/IP example, invoking our performance analysis technique for each configuration. Overall there were 24 points in this design space. Figure 9 shows the performance of the TCP/IP system when processing 10 packets of size 512 bytes under all possible combinations of priority assignments and DMA sizes. The best performance is obtained when the DMA size is 128 and priorities are assigned so that *Create_Packet*, *IP_Chk* and *Chksum* are in descending order of priority. The curves in the X-Y plane are iso- performance contours. The system performance is seen to vary between extremes of 2077 cycles and 3570 cycles. On a Sun Ultra10 Workstation, the entire design space exploration took less than 1 second of CPU time. The above experiment demonstrates that a) it is possible to perform thorough and fast exploration of the bus architecture design space using our technique and b) finding the ideal assignment of bus parameters that maximize performance of a given system is a very complex problem. For example, it may not be apparent why one priority assignment works better than another in the face of many synchronization events passing between the components. Though in the TCP/IP example increasing DMA size always benefits the system performance, it need not necessarily be so for systems in general, as we have seen in Section B.

VI. Conclusions and Future Work

Based on our investigation of the proposed performance analysis technique, we believe that its efficiency and accuracy would make it a useful addition to an SOC design environment. We believe that the proposed analysis framework can be applied to other bus architectures (e.g. hierarchical bus architectures and TDMA-based bus architectures [12]), and enhanced to also support performance analysis for other HW/SW communication mechanisms (in addition to bus-based and dedicated communication channel based). Further, we intend to integrate the developed performance analysis tool into a design exploration/optimization tool for SOC integration and communication architectures.

Acknowledgments. The authors would like to acknowledge Marcello Lajolo for his help on the POLIS and PTOLEMY frameworks.

References

[1] 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.

[2] J. A. Rowson and A. Sangiovanni-Vincentelli, “Interface Based Design ,” in *Proc. Design Automation Conf.*, pp. 178–183, June 1997.

[3] S. Bhattacharya, S. Dey, and F. Brglez, “Performance Analysis and optimization of schedules for conditional and loop-intensive specifications ,” in *Proc. Design Automation Conf.*, pp. 491–496, June 1994.

[4] 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.

[5] S. Malik, M. Martonosi, and Y. T. S. Li, “Static Timing Analysis of Embedded Software ,” in *Proc. Design Automation Conf.*, pp. 147–152, June 1997.

[6] R. Ernst and W. Ye, “Embedded program timing analysis based on path clustering and architecture classification,” in *Proc. Int. Conf. Computer-Aided Design*, pp. 598–604, Nov. 1997.

[7] 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.

[8] T. Yen and W. Wolf, “Communication synthesis for distributed embedded systems ,” in *Proc. Int. Conf. Computer-Aided Design*, pp. 288–294, Nov. 1995.

[9] 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.

[10] 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.

[11] M. Gasteier and M. Glesner, “Bus-based communication synthesis on system level ,” in *ACM Trans. Design Automation Electronic Systems*, pp. 1–11, Jan. 1999.

[12] VSI Alliance on-chip bus DWG, “On chip bus attributes specification, version v1.1.0.” (http://www.vsi.org/library/specs.htm).

[13] 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.

[14] 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.

[15] D. D. Gajski, F. Vahid, S. Narayan and J. Gong, *Specification and Design of Embedded Systems.* Prentice Hall, 1994.