

System-Level Performance Analysis for Designing On-Chip Communication Architectures

Kanishka Lahiri, *Student Member, IEEE*, Anand Raghunathan, *Senior Member, IEEE*, and Sujit Dey, *Member, IEEE*

Abstract—This paper presents a novel system-level performance analysis technique to support the design of custom communication architectures for system-on-chip integrated circuits. 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 in which an initial cosimulation 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 cosimulation containing necessary and sufficient information about the computations and communications of the system components. The system designer then specifies a communication architecture by: 1) selecting a topology consisting of dedicated as well as shared communication channels (shared buses) interconnected by bridges; 2) mapping the abstract communications to paths in the communication architecture; and 3) 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 1.88%) while being over two orders of magnitude faster.

Index Terms—Bus architectures, communication architectures, on-chip communication, performance analysis, simulation trace, system-on-chip.

I. 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 turnaround time. The ability to realize this potential depends on how well the designer exploits the configureability 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.

Manuscript received February 1, 2000; revised August 4, 2000. This work was supported by NEC USA Inc. and by the California MICRO program. This paper was recommended by Associate Editor R. Camposano.

K. Lahiri and S. Dey are with the Department of Electrical and Computer Engineering, University of California at San Diego, La Jolla, CA 92093 USA.

A. Raghunathan is with the Computers and Communications Research Laboratories, NEC USA, Princeton, NJ 08540 USA.

Publisher Item Identifier S 0278-0070(01)03539-4.

Achieving the disparate goals of reducing design turnaround time and thorough exploration of system-level tradeoffs requires efficient and accurate analysis tools to 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 need to be performed in order to derive a system architecture. The first step consists of partitioning and mapping parts of the system functionality into: 1) software that will execute on programmable processor(s); 2) parts that will be implemented by reusing (possibly adapting) existing function-specific cores; and 3) parts that will be compiled into hardware using hardware synthesis tools. In addition, the various components selected or assembled in the above step will require to share data and communicate in order to implement the system functionality. Therefore, the second step is to refine the communication requirements of the system into a communication architecture that best satisfies the communication needs of the components. While both of the above steps can significantly influence the quality of the resulting system architecture, in this paper, we focus on the latter. In particular, we address the issue of fast and accurate system-level performance analysis to drive the design of the communication architecture.

Recent efforts such as those of the Virtual Socket Interface Alliance (VSIA) are aimed at creating a design environment where a system designer can take advantage of experimenting with cores from multiple sources to choose the best combination for the target application. In particular, standardized bus interfaces for each “virtual component” [2] enables the design of novel bus architectures for application specific system-on-chips. With a potentially large number of choices in such a design environment, we believe it is increasingly important to provide a designer with automated support to evaluate and customize the on-chip communication architecture (both in terms of its topology and associated protocols) to best suit the needs of the application.

A. System-Level Performance Analysis

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 [3]–[8]. In these cases, techniques for automatic partitioning either: 1) ignore intercomponent communication entirely or 2) use simple models of communication to guide the partitioning/mapping step. Refining the abstract communications of the system into a specific communication architecture (with associated communication protocols) is performed as a subsequent step in the system design flow. In practice, while these two steps of system design are sometimes treated as

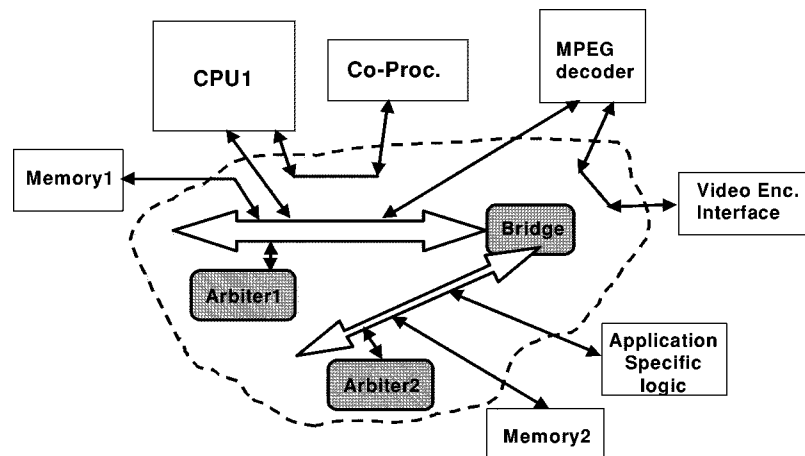


Fig. 1. System-on-chip components connected by a communication architecture.

separate problems for reasons of tractability [9], the merits of integrating communication protocol selection with hardware–software (HW/SW) partitioning are clearly demonstrated in [10]. Although our proposed technique currently does not support concurrent HW/SW partitioning and communication architecture exploration, it could be extended to do so by augmenting it with suitable existing techniques for HW/SW performance analysis.

While there is a large body of work focusing on the partitioning/mapping step, comparatively little research has addressed system-level performance analysis to help design high-performance communication architectures. Existing techniques that do consider the effects of the communication architecture can be broadly divided into the following categories.

- 1) Approaches based on simulation of the entire system using models of the components and their communication at different levels of abstraction [9], [11]. 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, limiting their computational efficiency.
- 2) Static system performance estimation techniques that include models of the communication time between components of the system [1], [10], [12]–[14]. 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, when they ignore dynamic effects such as waiting due to bus contention (e.g., [10] and [14]), or are overly pessimistic by assuming a worst-case scenario for bus contention (e.g., [12]).

B. System-on-Chip Communication Architectures

In this paper, we address performance analysis of systems such as the hypothetical one shown in Fig. 1, where components constituting the system-on-chip are integrated via a medium that allows exchange of data and control signals. This *communication architecture* could be as simple as a single *shared system bus*, or arbitrarily complex, consisting of a network of *shared*

and *dedicated* communication channels with some channels hierarchically connected by *bridges*.

Shared buses are very commonly used to facilitate communication between the various system components [2]. Being shared communication channels, they require arbitration (through a *bus arbiter*) in order to ensure that only one component (called a *bus master*) has control of the bus at any given time. Thus, a master that wishes to transfer data over the bus needs to first *handshake* with the arbiter in accordance with a fixed *bus protocol*. When multiple masters seek to use the bus simultaneously, the arbiter decides (typically based on a *priority scheme*) which one is granted the right to access the bus. In order to facilitate efficient transmission of larger chunks/bursts of data, bus protocols may also provide a direct memory access (DMA) or block transfer mode, wherein a master prenegotiates the right to use the bus for multiple bus cycles. In order to prevent any one master from monopolizing the bus and introducing large waiting for other access requests, a *maximum DMA block size* is typically placed on the amount of data that can be transmitted as a single DMA block.

Dedicated channels are point-to-point connections between components. Since such channels are not shared, the question of arbitration does not arise and communication on such channels can proceed as soon as both the sender and receiver are ready to communicate.

Bridges may exist between a pair of shared buses. The communication of data across a bridge requires a two step arbitration to take place. When the master and slave devices are located on two different buses, the master needs to negotiate with its local arbiter for access to its local bus. After this, the bridge negotiates with the arbiter of the remote bus (on which the slave is located) for access to the remote bus. Once these two grants are obtained, the data transfer can occur at a rate governed by the bus of lower bandwidth.

Each component that communicates with another must be physically connected to one or more than one (shared or dedicated) channels in the communication architecture. Furthermore, during execution, a component emits multiple *communication events*, each of which may vary in size from a single bit synchronization signal to a large data transfer consisting of multiple kilobytes. A single component may emit several

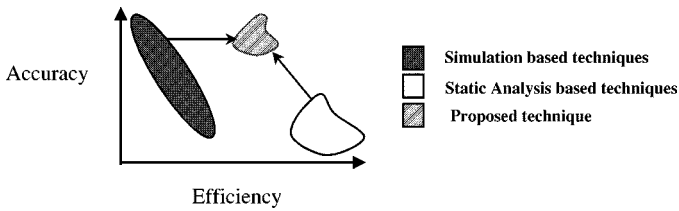


Fig. 2. Accuracy and efficiency of the proposed system performance-analysis technique relative to simulation-based and static-analysis-based approaches.

distinct communication events during its execution as well as multiple instances of the same event (e.g., a component executing a communication event within the body of a loop). Therefore, a specification of the communication architecture must not only include a mapping of components to channels, but a more fine-grained assignment of communication events to paths in the architecture.

C. Paper Contribution

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 to simulation-based approaches and static performance-analysis approaches is depicted in Fig. 2. 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 specification of a general communication architecture as described earlier. The designer is also allowed 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 interdependencies 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 cosimulation of the system is performed with the communication described in an abstract manner (as events or abstract data transfers). From this initial cosimulation, an abstract set of traces are extracted, containing necessary and sufficient information about the computations and communications of the system components. The system designer then specifies a communication architecture by: 1) selecting a topology consisting of dedicated as well as shared communication channels interconnected by bridges; 2) mapping the abstract communications to paths in the communication architecture; and 3) customizing the protocol used for each channel. The performance-analysis step involves extracting a communication analysis graph (CAG) from the initial cosimulation traces and subsequently operating on the CAG using information in the specification of the communication architecture. The result of manipulating the CAG is an estimate of the system performance for the given

communication architecture, as well as various useful statistics about the components and their communication.

The techniques presented in this paper are quite general since they enable us to study communication architectures that: 1) consist of an arbitrary interconnection of dedicated and shared communication channels and 2) have an arbitrary mapping of abstract communications to paths in the communication architecture.

We believe that our analysis technique will be useful in a computer-aided system design environment because, as we shall see in Section II, 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 [15], [16]. 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 cosimulation. This helps us to maintain accuracy comparable to complete system simulation on the one hand, while also achieving high efficiency by avoiding the problem of explosion of trace sizes. Since our analysis is trace-based, the results provided by our tool are specific to the input stimuli used for the initial cosimulation. As in the case of any simulation-based analysis, this places additional responsibility on the system designer to provide meaningful stimuli. However, given that cosimulation is the most popular system-level analysis technique in practice, we feel that this additional burden on the designer is quite reasonable.

II. MOTIVATION

In this section, we motivate the need for analysis techniques such as the one presented in this paper. We present a set of examples which demonstrate that the selection of a communication architecture that is well-suited to the communication traffic profiles of the application can significantly improve system performance. The examples presented in this section (along with others) are also used to conduct various experiments, whose results are presented in Section V. We model the problem of selecting a communication architecture as consisting of three steps: 1) the task of defining a communication *topology* that consists of a network of dedicated channels and shared buses, possibly connected by bridges; 2) the task of mapping the abstract communication events onto paths in the above topology; and 3) the task of selecting or customizing the protocol for each channel. We first illustrate these steps and their effects on system performance through examples.

A. Effect of Alternative Communication Architectures on System Performance

Example 1: We start by analyzing the performance of a part of a TCP/IP network interface system [17] under differing communication architectures. The subsystem consists

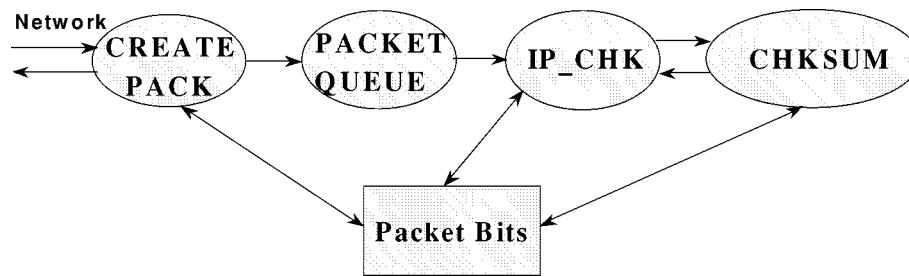


Fig. 3. TCP/IP network interface system.

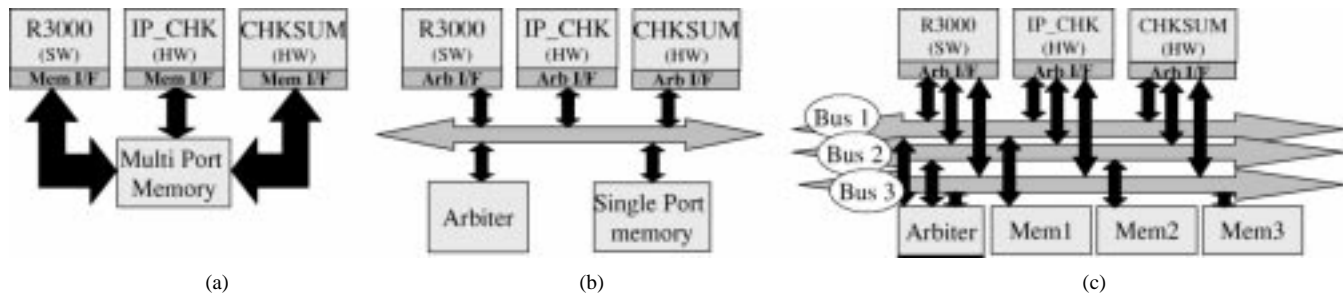


Fig. 4. Alternative communication architectures for the TCP/IP network interface system. (a) Dedicated links architecture. (b) Shared bus architecture. (c) Split-bus architecture.

of the part of the TCP/IP protocol related to the checksum computation performed at a network interface card (Fig. 3). For incoming packets, the task *Create_Pack* 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 and the number of bytes to the *Packet_Queue*. From this queue, *IP_Chk* retrieves a new packet, overwrites parts of the header (which should not be used in the checksum computation) with zeros, and signals to the *Chksum* task that a new packet 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. Fig. 4(a)–(c) shows one partitioning and mapping, where the tasks *Create_Pack* 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. Fig. 4(a) shows a candidate communication architecture where the system components access a shared multiport memory through dedicated links. This communication 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_Pack* can be working on packet $i + 2$, all at the same time. At any given time, 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 Fig. 4(b). Here, the components of the system access a shared single-port memory

through a common system bus. In this shared bus communication architecture, an arbiter resolves conflicts resulting from simultaneous attempts to access the bus.

We performed experiments to observe the performance of the TCP/IP system under various memory and communication architectures using POLIS [18] as a HW/SW codesign tool and PTOLEMY [19] for system-level simulation. We used a behavioral bus arbiter model [17] to take into account the effect of the shared bus communication architecture on system performance.

Our experimental results show that the processing time per packet of each component for the shared bus architecture of Fig. 4(b) is up to 40% higher than that for the dedicated link case of Fig. 4(a). The degradation is because the shared bus introduces waiting time whenever two components simultaneously request access to memory, whereas in a dedicated link architecture, the components are permitted to concurrently access memory.

The next communication architecture considered is one based on a split bus. Fig. 4(c) shows this alternative implementation of the TCP/IP subsystem, where the shared 128-bit bus is split into three 32-bit busses, each connected to a separate memory component. In this new configuration, packet i is stored in *Mem1* and can be processed by the tasks using *Bus1*, while packet $i + 1$ can be stored in *Mem2* and be processed by the tasks using *Bus2*, without generating any bus conflicts. However, in the split-bus implementation, each bus now has reduced bandwidth, leading to a possible degradation in performance compared to a shared bus implementation. This leads to a tradeoff between the higher bandwidth of a shared bus and the lower frequency of conflicts in a split-bus configuration.

We conducted experiments to evaluate the three-way 32-bit split-bus architecture versus the 128-bit shared bus architecture for packets of varying size. The results show potentially significant performance improvements under the split-bus communication architecture under certain situations. For example, a 44%

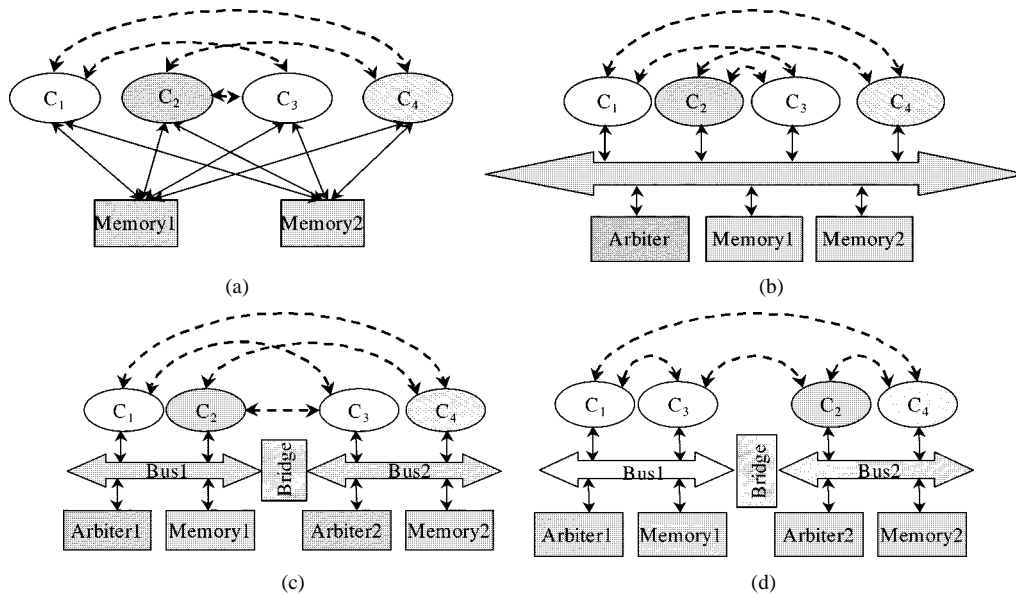


Fig. 5. Alternative communication architectures for an example system. (a) MEM4: abstract communication. (b) MEM4: single shared bus. (c) MEM4: multiple buses. (d) MEM4: multiple buses, different mapping.

reduction in execution cycles per packet is obtained when the size of the input packets is 16 B. However, the results also show that a split-bus communication architecture does not always give superior performance. For instance, when the size of the input packets is 512 B, the shared bus architecture has about a 20% better performance than the split-bus implementation, owing to the reduced bandwidth in the split-bus case. ■

Example 2: Consider the system described in Fig. 5(a), which 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 synchronization¹ (represented by dotted arcs) and data transfer (represented by solid arcs). Fig. 5(b)–(d) shows three alternative topologies for the system communication architecture. In Fig. 5(b), a single shared bus is used to implement all data transfers, whereas in Fig. 5(c) and (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 Fig. 5(b), all data transfers to and from the shared memories are mapped to the only bus, while the intercomponent synchronization is implemented using dedicated communication channels. In Fig. 5(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 Fig. 5(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 Fig. 5(b)–(d) will result in the best overall system performance. These include the following.

- 1) 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.
- 2) 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 Fig. 5(c), when component C_1 needs to access $Memory2$, it first makes a request to $Arbiter1$ 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.
- 3) 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 components are good candidates to share the same communication channel. Components

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

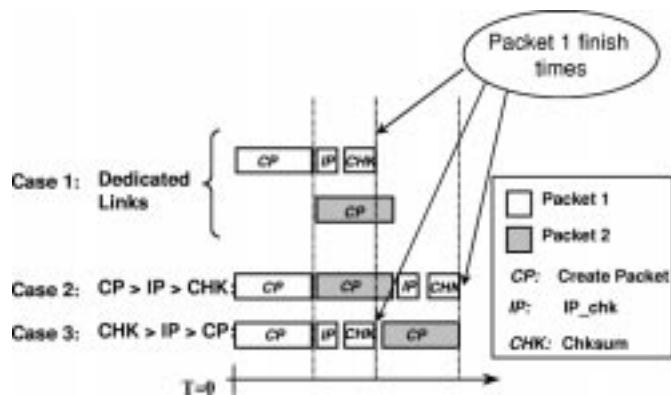


Fig. 6. Variation of processing time with priority assignments.

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 tradeoffs. For example, consideration of the potential parallelism when using multiple buses suggests that the architectures of Fig. 5(c) and (d) are superior to the architecture of Fig. 5(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 Fig. 5(b)–(d) were 224 031 cycles, 244 002 cycles, and 165 987 cycles, respectively. Thus, the single bus architecture of Fig. 5(b) is actually 9% *faster than* the two bus architecture of Fig. 5(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 Fig. 5(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 improvements due to parallelism.

The communication architecture of Fig. 5(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 intercomponent synchronization). As a result, the communication architecture of Fig. 5(d) also results in fewer bus conflicts for each bus compared to the architecture of Fig. 5(c). ■

B. Effect of Customizing Bus Protocols on System Performance

Our next experiments show 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.

Example 3: Consider the execution traces of two packets by the TCP/IP tasks *Create_Pack*, *IP_Chk*, and *Checksum* under three different cases, as shown in Fig. 6. Case 1 reflects the case

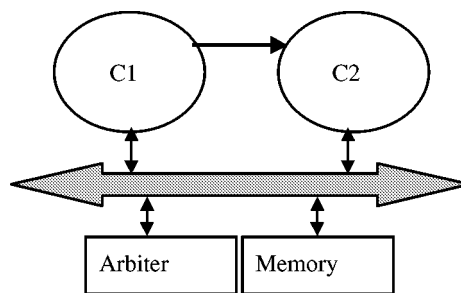


Fig. 7. Example system to illustrate effect of DMA size on performance.

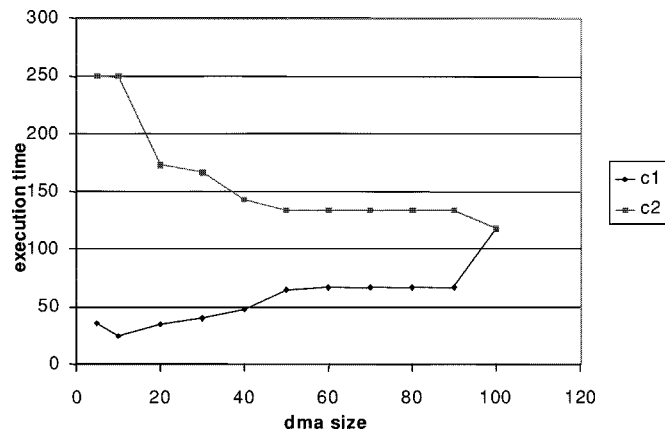


Fig. 8. Effect of DMA size on performance.

when the second packet can be processed by *Create_Pack* while *Checksum* processes the first, without any conflict. This execution trace can be generated by using a dedicated link architecture like Fig. 4(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. Fig. 6 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 versus shared), but also the task priorities used. ■

Example 4: Consider a simple system shown in Fig. 7 consisting 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 ten cycles and memory transfers of average size ten bus words, while C_2 performs computations of average size ten cycles, but memory transfers of average size 100 bus words. Fig. 8 shows the effect of varying DMA sizes (x axis) on system performance (y axis). We observe the following.

- 1) The choice of bus parameters like DMA size can significantly affect system performance. For example, Fig. 8

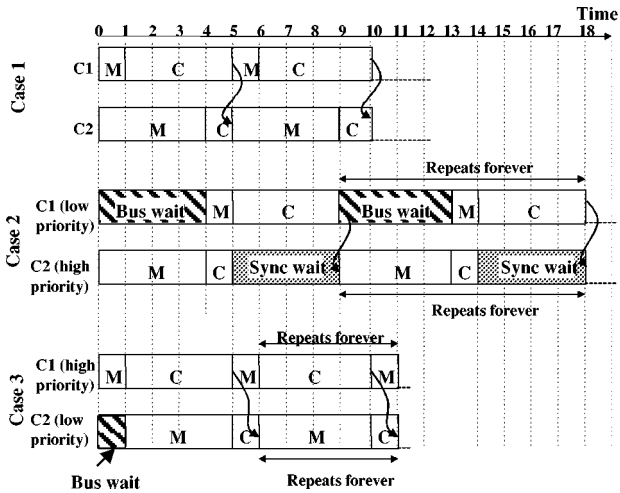


Fig. 9. Indirect effect of bus architecture on synchronization wait time.

shows that the performance range for $C2$ for varying DMA sizes is 117–250 clock cycles.

- 2) 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 (decreases execution time) of $C2$, it has a negative effect on $C1$, whose computation and bus access profile is different from that of $C2$. ■

Example 5: Consider again the two component system shown in Fig. 7. Recall that components $C1$ and $C2$ access a global memory through a shared bus and synchronize with each other. The traces shown in Fig. 9 represent the operation of the system of Fig. 7 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) was 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) was 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 intercomponent synchronization wait time has also been eliminated.

This example illustrates the importance of considering indirect effects of the bus architecture on the timing of the various system components. It shows that the computation, synchro-

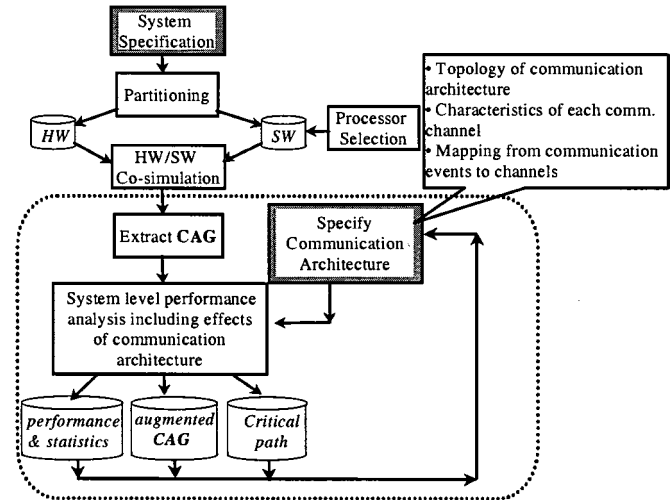


Fig. 10. Two-phase performance-analysis methodology.

nization, and communication times of various components in a system are *interdependent*. Thus, a separate analysis of the communication time alone will not necessarily reflect the total system performance accurately. ■

The above investigations demonstrate the criticality of selecting the optimal communication architectures and bus protocols, and thereby the need for fast and accurate performance-analysis techniques that can evaluate the numerous choices. The last example demonstrates that merely considering the direct effects of the communication architecture without considering the indirect effects on the timing of the system could lead to erroneous performance estimates. As mentioned in Section I, static analysis techniques used for estimating the communication time in previous research are not accurate enough to drive the design of such communication architectures, while a complete HW/SW cosimulation of the system is too time consuming to perform iteratively in a design exploration framework.

III. 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 Fig. 10.

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 communication architecture. Here, communication is modeled at an abstract level by the exchange of events or tokens. The time taken to generate and consume a communication event depends only on the size of the data transfer associated with it and not on the number of concurrent communications. Hence, the output of this step is a *timing inaccurate* system execution trace.

The second phase (enclosed within the dotted box in Fig. 10) performs system performance analysis including the effects of the selected communication architecture. To do this, from the trace obtained in the first phase, we construct a CAG, which captures the computations, communications, and the synchronizations seen during simulation of the entire system. In addition, the designer specifies the communication architecture to implement

the various communications. 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:

- 1) an augmented version of the CAG, which has incorporated various latencies introduced as a result of moving from an abstract communication model to a refined one;
- 2) an estimate of the performance of the entire system;
- 3) the system critical path (or paths) as a sequence of computations and communications;
- 4) numerous execution statistics regarding bus usage, conflicts, the proportion of the system critical path occupied by each component, etc.

Using this two-phase methodology, the time consuming pre-processing step need not be performed iteratively. Alternative communication architectures can be evaluated within the second phase of the methodology. Being fast and accurate (for reasons discussed later in this section), 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 Fig. 10, the second phase consists of three steps: 1) abstracting information from the simulation trace and constructing the CAG; 2) specifying the communication architecture; and 3) analyzing the system performance under the given communication architecture. In Section III-A and Section III-B, we discuss each of these steps in turn and present details of the analysis method in Section IV.

A. Extracting Information from HW/SW Cosimulation

Simulation traces obtained via HW/SW cosimulation can be very complex and detailed, containing a large number of computations and communications. A system component emits a *communication event* each time it executes a data transfer or synchronization signal in its mapped specification. Correspondingly, it emits *computation events* that indicate the various operations that the system component performs throughout the course of its execution. Since the system specification typically contains loops, the execution trace will contain multiple instances of any computation or communication event. The total number of events in the trace, therefore, directly depends on the length of the simulation and the complexity of each system component.

Using detailed traces for our analysis is undesirable because: 1) extensive traces are required to capture all the representative execution paths of a system and 2) system components can be very complex, e.g., a CPU executing several processes. To avoid the high complexity of analyzing a potentially large number of simulation events, in our approach, we extract from the traces only information 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 events and

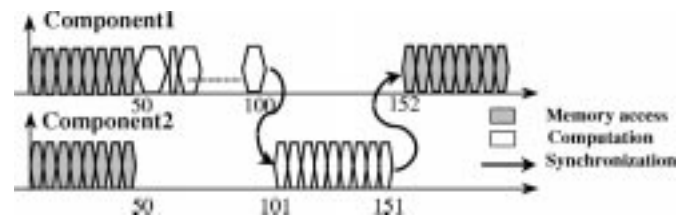


Fig. 11. Traces generated by HW/SW cosimulation.

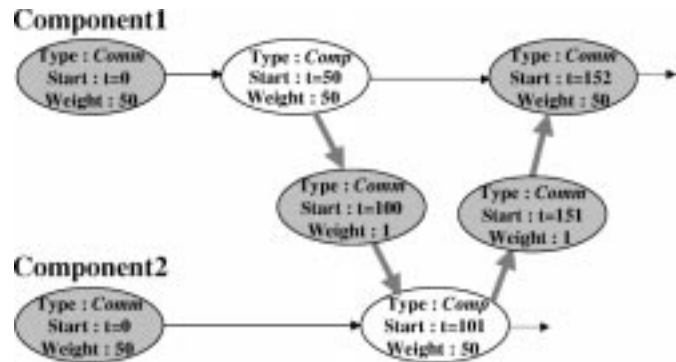


Fig. 12. CAG generated from traces in Fig. 11.

communication events into abstract computation and communication clusters. This involves identifying the start and end points (time-stamps) of a stream of data transfers from a component (multiple instances of the same communication event), and replacing all the constituent events by a single communication vertex. The vertex is assigned an *event id*, which is derived from the event identifier of the constituent communication events. The same approach is used for contiguous computation events, whereby multiple computations in the execution trace are replaced by a single computation vertex. For example, in the traces shown in Fig. 11, *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, but construct abstract vertices as described above. Next, these vertices are collected into a CAG. Fig. 12 shows a simple CAG that represents the traces of Fig. 11. The graph has two kinds of vertices—*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 intercomponent 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 each vertex in the CAG represents a large number of simulation events.

B. Definition of the Communication Architecture

The analysis technique supports communication architectures with the following characteristics:

- 1) on-chip communication using channels, where each channel may either be a dedicated link or a shared bus;

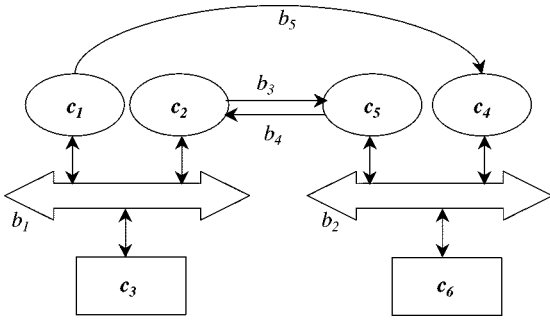


Fig. 13. Communication architecture involving shared buses and dedicated links.

TABLE I
DESCRIPTION OF COMMUNICATION CHANNELS

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

- 2) parameterized characterization of each channel;
- 3) arbitrary mapping of events to channels.

We next briefly describe the characteristics that constitute a candidate communication architecture. Consider the system shown in Fig. 13 consisting of six 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, as well as a mapping of communication events to channels.

Table I shows a set of parameters for each channel. For example, for channel b_1 , $width = 32$ (in bits), $speed = 66$ MHz, 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 II, 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 II as a *dedicated link*. Since more than one component in Fig. 13 may simultaneously try to use b_1 for communication, 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 ten. Table II 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, 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.

TABLE II
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 $e_4 \rightarrow b_1$
...	...

C. Implementation

In our implementation of the design flow described in Fig. 10, partitioning, processor selection, and HW/SW cosimulation is performed using the POLIS [18] and PTOLEMY [19] HW/SW codesign environment. A set of observation ports are created in the PTOLEMY system-level netlist to trace execution of specific events. During cosimulation, time-stamped events seen at these ports are gathered as an abstract simulation trace. This trace is then translated into a description that is independent of PTOLEMY specifics, yet is readable by our analysis tool. This will facilitate porting our analysis tool to other HW/SW codesign environments.

The initial CAG is constructed by making a single pass of the simulation trace. Other inputs (Tables I and II) are used to guide the execution of the analysis tool on the extracted CAG. The tool suitably manipulates the CAG and generates a report of the system performance under the chosen communication architecture. The output includes an augmented version of the CAG, which has incorporated various latencies arising out of the specification of a detailed communication architecture. Other than this, the tool calculates the performance of the system, the system critical path, statistics regarding conflicts seen on each shared bus, and the proportion of the critical path occupied by each component.

D. Accuracy and Efficiency Issues

The efficiency of our performance analysis tool is derived from the fact that we abstract out the details of the computations and communications between the system components and cluster them into vertices while constructing the CAG. 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 communication vertex, we only care about the amount of data transferred. As a result, a CAG 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 CAG is constructed from the dynamic traces resulting from cosimulation, i.e., it represents the execution of the system *unrolled in time*. Overall, the computational complexity of our performance-analysis technique is *linear in the size of the CAG* and, hence, is much faster than complete system cosimulation. The efficiency of our performance-analysis technique is further borne out by the experimental results presented in Section V.

The accuracy of our performance analysis technique is due to two factors.

- 1) Since we are using a dynamic execution trace derived from cosimulation, the control flow within each component is fully determined (e.g., we do not need to worry about predicting how many times each loop is executed, how each branch is taken, etc.).
- 2) 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 communication architecture, the importance of which was demonstrated in Section II.

It bears mentioning at this point that we assume that the actual operations performed in the computation vertex and the data transferred in the memory access vertices are not dependent on the bus effects. Put in a different way, the communication 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 [3], [18].

IV. ALGORITHMS

In this section, we describe the various algorithms that are executed by the performance-analysis tool. First, we address performance analysis assuming the communication architecture consists of a single system-wide shared bus, like that of Fig. 4(b). Therefore, all communication vertices (including synchronizations and data transfers) in the initial CAG are mapped to the shared bus. We then show how this algorithm can be easily extended to support a general topology of shared buses and dedicated channels with arbitrary mapping of components and emitted communication events to channels. Last, we show how we can also take into account the effect of bridges in the communication architecture that allow communication paths between two components to span multiple channels.

A. Analysis of a Single Shared Bus

First, we describe through examples the effect of the execution of our algorithm on portions of an extracted CAG, assuming every communication vertex in the CAG is mapped to a single shared bus.

Fig. 14(a) shows a portion of a CAG containing a single communication vertex representing an instance of communication *event#6* emitted by *Comp1*. The bold arrows indicate control flow arising out of sequential execution of computations and communications in *Comp1*, while the rest indicate control dependencies arising out of intercomponent synchronization. Since communication *event#6* is mapped to a shared bus, we need to take into account the intrinsic overhead of the bus protocol (apart from dynamic effects like bus contention) that is introduced while this instance of communication is executed. To do this, a new vertex of type *HANDSHAKE* (HS) is introduced in the CAG. The start time of the new vertex is 230, same as that of the original communication vertex. The weight of five cycles is derived from the bus protocol specification and the actual start

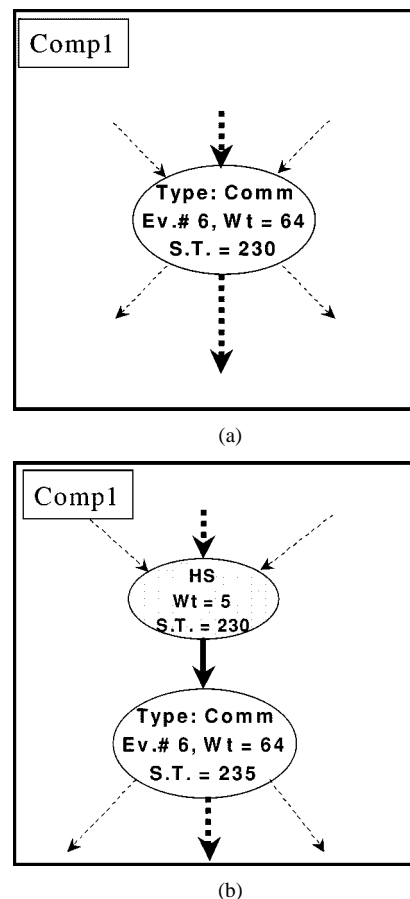


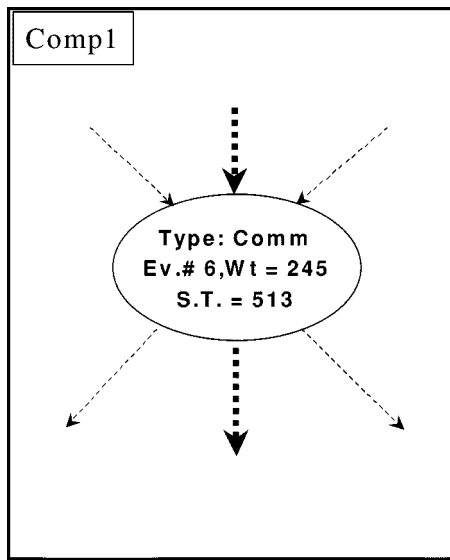
Fig. 14. Effect of overhead of the bus protocol—handshaking. (a) Portion of initial CAG. (b) Portion of transformed CAG.

time 235 of the communication vertex is recalculated based on this weight. Fig. 14(b) shows the CAG after this transformation.

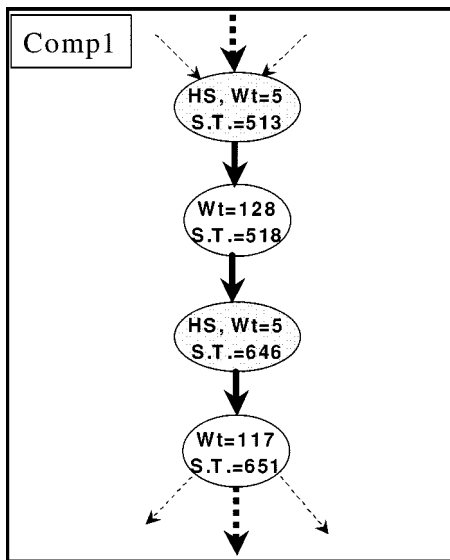
Fig. 15(a) shows a communication vertex, also mapped to a shared bus, that exceeds the maximum permissible DMA/burst transfer size for the shared bus. Fig. 15(b) shows the effect of taking this into account during performance analysis, where additional vertices of weight 128 and 117 have been created to restrict the size of each bus access to the maximum DMA size of 128 cycles (assuming one bus word per cycle). Also, appropriate *HANDSHAKE* vertices have been introduced.

Fig. 16 shows a portion of the CAG involving execution sequences of two components. Vertices v_1 and v_2 are communication vertices that belong to two different components and are activated at the same time. Since the bus is shared, this represents a bus conflict and one of the components must be made to wait. The result of the transformation is shown in Fig. 16(b). The analysis tool examines the two contesting components and “grants” bus access to the one of higher priority, say *Comp1*, by examining the specification of the bus protocol. Consequently, *Comp2* must wait and the start times on its subsequent vertices are modified to reflect this delay. Also, an additional dependency (shown by a dotted arrow) is introduced to represent the mutually exclusive nature of access to the shared bus.

The algorithm executed by the analysis tool (assuming a single shared bus) is shown in Fig. 17. It traverses the entire CAG, modifies it to account for the effects of the shared bus,



(a)



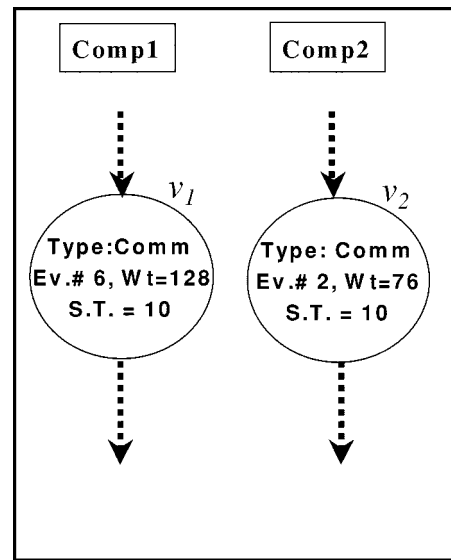
(b)

Fig. 15. Effect of bus protocol—DMA size. (a) Portion of initial CAG. (b) Portion of transformed CAG.

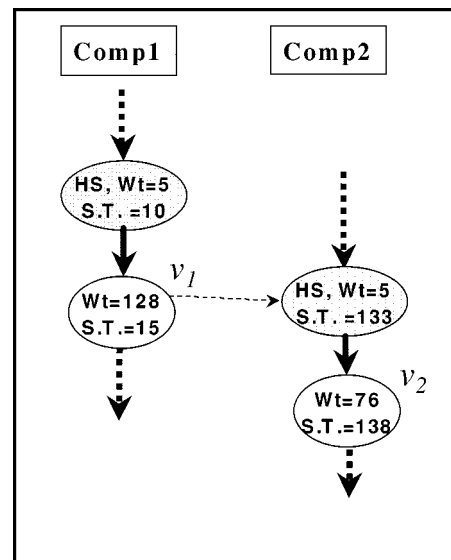
and assigns start times to each vertex in the graph that reflect its actual execution time.

First, the algorithm reads the details of the bus protocol into a structure called *bus*. *Ready_list* is a set of vertices all of whose predecessors have been executed. It contains a subset of the vertices in the CAG and is sorted on the basis of start time and priority. It is initialized to contain those vertices in the CAG that have no predecessors. The algorithm proceeds by invoking *dequeue*, which removes the vertex at the top of *ready_list*. If v is a *computation* vertex, it is executed. The function *execute*(v) involves removing v from *ready_list* and marking it *visited*. Housekeeping operations involve adjusting the *start_times* of its successors (*modify_successor_start_times*) and inserting any enabled vertices into the *ready_list* (*add_enabled_vertices*).

If v is a *communication* vertex, its actual size in terms of bus cycles is calculated from the width and speed of the bus and the original weight of the vertex. While the weight of a communi-



(a)



(b)

Fig. 16. Effect of bus protocol—conflicts and priorities. (a) Portion of initial CAG. (b) Portion of transformed CAG.

cation vertex in the initial CAG is calculated from the size of the data transfer (in bytes) and an assumed fixed data transfer rate, the algorithm scales the weight in proportion to the width and speed of the channel.

The *start_time* t_1 of the dequeued vertex v represents the time at which it makes the request for access to the bus. A check is made for the most recent vertex that accessed the bus. If it has a *finish_time* of t_2 (where $t_2 > t_1$), the tool will delay the *start_time* of the requesting vertex v to time t_2 .

A *handshake* vertex is constructed to take into account the overhead of the protocol. If the size of the dequeued vertex v is larger than the DMA size, a new vertex v' of size equal to the DMA size is created, inserted into the graph, and executed. The *start_time* and *weight* of the original vertex v are modified and reinserted into the *ready_list*. Waiting for the bus and resizing of vertices may change the *finish_time* of the currently executing vertex v . Therefore, its new *start_time* plus its new weight is

```

inputs: CAG G, Bus Protocol Description
outputs: CAG G, Performance Statistics, Critical Path
initial
  mark all vertices visited = false;
  read_Protocol(bus);
  initialize(ready_list, G);
begin
  do
    v = deque(ready_list);
    if v.type = COMP
      execute(v);
      modify_successor_start_times(v);
      add_enabled_vertices(v);
    elseif v.type = COMM
      v.weight = get_weight(bus.width, bus.freq);
      if bus.next_free_time > v.start_time
        v.start_time = bus.next_free_time;
      endif
      create(w, HS);
      w.start_time = v.start_time;
      w.weight = bus.protocol_overhead;
      insert(w, G);
      execute(w);
      v.start_time = w.finish_time;
      if v.weight < bus.DMA_SIZE
        execute(v);
        bus.next_free_time = v.finish_time;
        modify_successor_start_times(v);
        add_enabled_vertices(v);
      else
        create(v', COMM);
        v'.start_time = v.start_time;
        v'.weight = bus.DMA_SIZE;
        insert(v', G);
        execute(v');
        v.start_time = v'.finish_time;
        v.weight = v.weight - channel[i].DMA_SIZE;
        insert(v, ready_list);
      endif
    endif
  until no more ready vertices
  generate_stats();
end

```

Fig. 17. BusAnalyzer algorithm.

used to update the *start_times* of its successors. This is done by *modify_successor_start_times*. Enabled successors are then added to *ready_list* by *add_enabled_vertices*. This function also accumulates statistics that help determine the critical path.

The output of the algorithm includes execution statistics, performance, and critical path information. The algorithm terminates when all the vertices of the CAG are visited exactly once.

B. Extension to Arbitrary Communication Topologies

To allow this extension, we need to enhance the algorithm of Fig. 17 to the one shown in Fig. 18. The database *channel*, derived from a specification of the communication architecture, contains information about each channel in the architecture. Channels are numbered 1 through MAX_CHANNELS and details of the channel b_i are stored in $channel[i]$. *Ready_list* is now a set of lists, one for each channel in the communication ar-

```

inputs: CAG G, Communication Arch Description
outputs: CAG G, Performance Statistics, Critical Path
initial
  mark all vertices visited = false;
  read_Architecture(channel[MAX_CHANNELS]);
  initialize(ready_list[MAX_CHANNELS], G);
begin
  do
    i := get_channel(ready_list);
    v := deque(ready_list[i]);
    if v.type = COMP
      execute(v);
      modify_successor_start_times(v);
      add_enabled_vertices(v);
    elseif v.type = COMM
      get_weight(v.weight, channel[i]);
      if channel[i].type = shared
        execute_bus_protocol(channel[i]);
      if channel[i].type = dedicated
        execute(v);
        modify_successor_start_times(v);
        add_enabled_vertices(v);
      endif
    endif
  until no more ready vertices
  generate_stats();
end

```

Fig. 18. CommAnalyzer algorithm.

chitecture. The first list $ready_list[0]$ is a special list containing ready computation vertices while $ready_list[i]$, ($i > 0$) contains the set of vertices that represent ready communications mapped to channel b_i .

The algorithm proceeds by invoking *get_channel*, which examines the first vertex of every list and chooses to execute the one with the earliest *start_time*, say, *v*. If *v* is a computation vertex, it is executed (as before). If it is a communication vertex dequeued from $ready_list[i]$, it is first appropriately scaled using information about channel b_i derived from the *channel* database. Further, if channel b_i is a shared channel, the associated bus protocol is imposed on it to account for protocol overhead, priorities, and DMA size. If channel b_i is not a shared channel, *v* is executed just like a computation vertex.

C. Extension to Include Effect of Bridges

It is possible that two buses in the communication architecture are connected by a bridge to allow communication between components connected to different buses. We next show how our performance-analysis technique takes into account possible presence of such structures in the communication architecture.

Fig. 19(a) shows a portion of the initial CAG, where a communication vertex of size 128 (an intercomponent data transfer from C_2 to C_5) has been mapped to the path b_1 - b_2 , (a pair of buses) connected by a bridge shown in Fig. 13. The mapping is described as part of the specification of the communication architecture as in Table II. In this case, *event#31* is mapped to the path b_1 - b_2 , instead of a single bus.

In such a situation, the data transfer occurs over three stages. First, C_2 secures b_1 by negotiating with its local arbiter. Second, the bridge negotiates with the arbiter of b_2 to secure bus b_2 , and

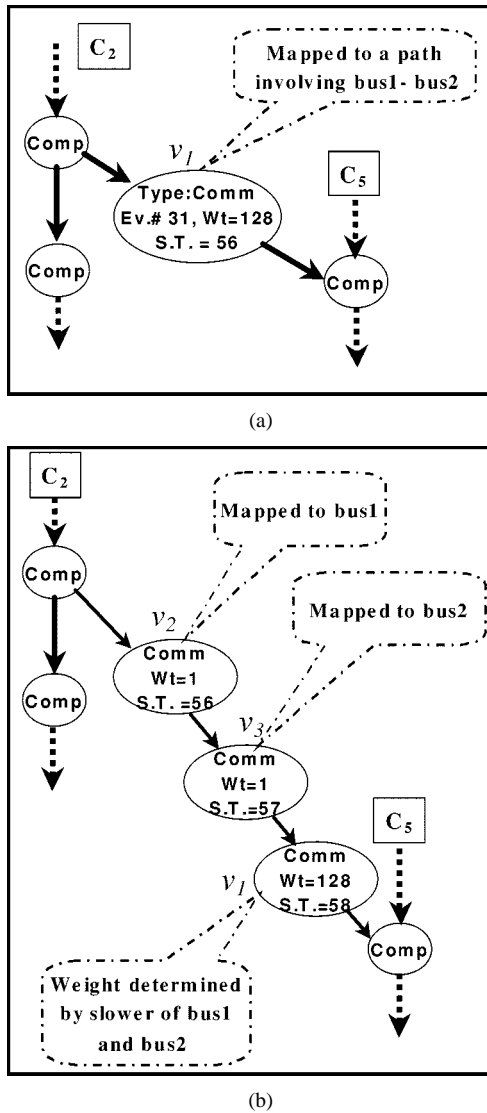


Fig. 19. Effect of bridges. (a) Portion of initial CAG. (b) Portion of transformed CAG.

finally, the data gets transferred at the speed of the slower bus. Thus, there are the effects of two bus protocols to be taken into account. The transformed CAG is shown in Fig. 19(b), where two additional vertices mapped to the two buses have been created to take into account the two step arbitration process. For the first arbitration, C_2 plays the role of the bus master (on b_1) and the bridge the slave, while in the second arbitration, the bridge plays the role of the bus master (on b_2), while C_5 is the slave.

Additionally, no other ready vertex should prevent v_1 from executing once v_2 and v_3 have executed. This can be done very simply by boosting the priority of v_1 to ensure it is at the top of the *ready_list* as soon as it is ready to execute. These enhancements and others pertaining to collecting execution statistics are easily integrated into the algorithms of Figs. 17 and 18.

V. EXPERIMENTAL RESULTS

In this section, we demonstrate through experiments the accuracy and efficiency of our technique and its consequent utility in

exploring the design space of communication architectures for various example systems.

We used four example systems in these experiments—the TCP/IP network interface card subsystem of Fig. 4, two systems that are similar to the two component system of Fig. 7, and the four component system shown in Fig. 5. For each system, we performed two experiments. In the first experiment, we used a complete system cosimulation in the POLIS/PTOLEMY framework using behavioral models of the communication architecture [17]. All system components were specified using Esterel and C while graphical schematic capture was performed in PTOLEMY. In the second experiment, we used the proposed performance-analysis technique to estimate the total system performance. This was done by collecting traces from an initial cosimulation (with no explicit modeling of the communication architecture), converting them into an equivalent CAG, and executing our analysis tool, providing it with a description of the communication architecture to be evaluated.

To confirm the accuracy and efficiency of our technique, the system performance and running time measurements obtained in the second experiment are compared against those measured in the first, namely, complete cosimulation of the system.

A. Experiments with Shared Bus Protocols

In the first set of experiments, we examine the effectiveness of our analysis tool on two example systems where all communication occurs across a single system bus and we show how it can be used to investigate alternative bus protocols.

The systems we choose are the TCP/IP network interface card subsystem and variants of the two component system in Fig. 7. The latter two systems are structurally equivalent, but differ from each other significantly in the computation and bus access profiles exhibited by each component. For each system, an arbitrary set of values were chosen for the parameters of the bus protocol. For the TCP/IP system we simulated 100 packets, each of size 512 B with the bus parameters chosen as follows: *MAX_DMA_SIZE* = 16 bus words, *PROTOCOL_OVERHEAD* = 1 cycle, *BUS_WIDTH* = 8 bytes, speed = 166 MHz. The priorities were set so that *Create_Pack* has the highest priority followed by *IP_Chk* followed by *Chksum*. In the second system (*SYS1*), components C_1 and C_2 access the bus for an average of ten words at a time and execute computations of average duration ten cycles. DMA block size was set to five, priorities arbitrarily assigned, and the system was studied for an execution period containing 2000 memory accesses from each component. The remaining parameters were set to the same values as in the TCP/IP system. In the third system, (*SYS2*), C_1 has an average bus access of duration 100 bus words while C_2 has an average bus access of size ten. The system was studied for 2000 iterations of C_1 and 400 iterations of C_2 .

Tables III and IV present the results of our experiments. Table III reports the performance estimates obtained by complete system cosimulation (second column), the performance estimate obtained using our analysis technique (third column), and the percentage difference between the two (fourth column). Table IV reports the efficiency (execution time) of a complete

TABLE III

ACCURACY OF THE ANALYSIS TECHNIQUE FOR SHARED BUS ARCHITECTURES

Example System	Co-simulation estimate (cycles)	CAG Analysis Estimate (cycles)	% variation
TCP/IP	22877	22997	0.05
SYS1	68146	67827	0.47
SYS2	69858	71400	2.21

TABLE IV

EFFICIENCY OF THE ANALYSIS TECHNIQUE FOR SHARED BUS ARCHITECTURES

Example System	Co-simulation Elapsed time (sec)	CAG graph Elapsed time(sec)
TCP/IP	87	0.05
SYS1	922	0.22
SYS2	638	0.13

system cosimulation (second column), as well as our performance-analysis technique (third column). (Measurements of elapsed time were made on a Sun Ultra-10 workstation with 128-MB RAM running Solaris 2.6.)

The results of Table III indicate that our technique has a negligible loss of accuracy compared to complete HW/SW cosimulation. 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 IV shows that our performance-analysis technique is two to three orders of magnitude faster than complete HW/SW cosimulation.

In order to demonstrate the utility of our performance-analysis technique in an iterative design space exploration framework for a shared bus protocol, 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 36 points in this design space. Fig. 20 shows the performance of the TCP/IP system when processing ten packets of size 512 B 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_Pack*, *IP_Chk*, and *Checksum* 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. The entire design space exploration experiment took less than 1 s of CPU time.

The above experiment demonstrates that: 1) it is possible to perform thorough and fast exploration of the bus protocol design space using our technique and 2) 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 II-B.

Performance vs. Priority and DMA size for TCP/IP subsystem

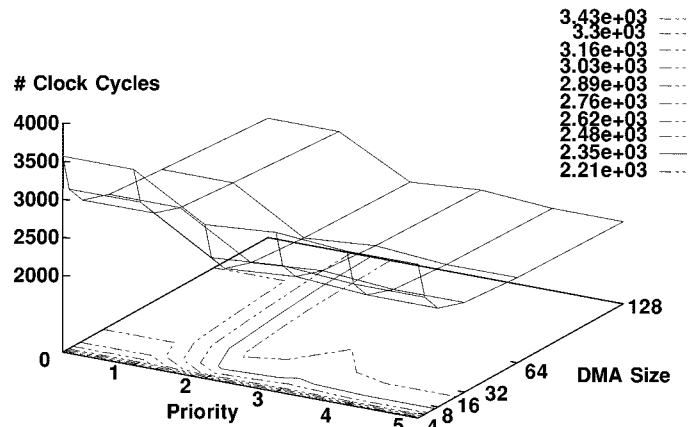


Fig. 20. Efficient design space exploration for the TCP/IP system using the proposed estimation technique.

TABLE V
COMPARISON OF ALTERNATIVE ARCHITECTURES OF MEM4

MEM4 Communication Architecture	CAG Analysis Estimate (cycles)	Co-simulation Estimate (cycles)	Error %
Config = Case 1, Bus ₁ :width = 8, DMA = 50, C1>C2>C3>C4, Latency = 1	224031	231970	-3.42
Config = Case 2, Bus ₁ :width = 8, DMA = 50, C1>C2, Latency = 1, Bus ₂ :width = 8, DMA = 50, C3>C4, Latency = 1	244002	245696	-0.70
Config = Case 3, Bus ₁ :width = 8, DMA = 50, C1>C3, Latency = 1, Bus ₂ :width = 8, DMA = 50, C2>C4, Latency = 1	165987	169999	-2.36

B. Experiments with General Communication Architectures

The next set of experiments demonstrate the effectiveness of our analysis technique while investigating alternative communication architectures, with multiple communication channels connected by bridges, and arbitrary mapping of events to channels. We also show how interesting tradeoffs can be closely examined while exploring this large design space.

Experiments were conducted on MEM4, the four component system shown in Fig. 5(a) and described in Section II. Table V reports the results of performance analysis using both the proposed CAG-based analysis method and the traditional approach of cosimulation on three alternative communication architectures, Case 1 to Case 3 shown in Figs. 5(b)–(d), respectively.

The results in Table V demonstrate the following.

- 1) The analysis technique is accurate even when estimating performance of the system given arbitrary communication architectures—the estimates obtained are within 3.5% of those obtained via cosimulation.
- 2) 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

TABLE VI
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 ₁ = bus ₂ = bus ₃ : width = 32, DMA_SIZE = inf., CREATE_PACKET>>IP_CHK>> CHKSUM, Latency = 1	35079	36075	-2.76
Config = Case 2, bus ₁ = bus ₂ = bus ₃ : width = 16, DMA_SIZE = inf., CREATE_PACKET>>IP_CHK>> CHKSUM, Latency = 1	67645	66636	1.51

TABLE VII
COMPARISON OF RUNNING TIME OF CAG BASED ANALYSIS AGAINST COSIMULATION

Case Study	Co-simulation for CAG Generation (seconds)	CAG Based Analysis (seconds)	System co-simulation (seconds)	Speedup
MEM4 in Case1 configuration	220	8.2	1325	162
MEM4 in Case2 configuration	0	8.3	1895	228
MEM4 in Case 3 configuration	0	7.6	1863	245
TCP_IP in Case 1 configuration	84	3.0	688	229

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%.

We next conducted experiments on alternative architectures for the TCP/IP network interface card system involving multiple buses. Two alternative communication architectures are shown: Fig. 4(b), using a shared bus (Case 1), and Fig. 4(c), using three buses (Case 2). Table VI shows results of simulating the system under one configuration of Case 1 and for two configurations of Case 2.

Again, it is clear that no loss of accuracy has been suffered. Additionally, Table VI 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 three buses that allows 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 countered by the price of lower bandwidth on each bus.

Table VII compares the efficiency of our technique versus complete system cosimulation for four of the configurations studied. In the first column of the table, we additionally report the time taken by the first phase (cosimulation 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 cosimulated, including behavioral hardware models of the communication architecture. Note that system cosimulation using the hardware models of the communication architecture (third column of Table VII) takes significantly more time than cosimulation performed with an abstract model of communication (second column of Table VII).

In row three, it is observed that the speed up of using our analysis over conventional HW/SW cosimulation is 245X. In general, Table VII reports a speedup of two orders of magnitude for each system investigated. Note that the first column entries of rows 2 and 3 of Table VII are zero because only one initial cosimulation (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 cosimulation approach (fourth column).

VI. CONCLUSION

In this paper, we have demonstrated that the selection of the communication architecture for a system (its topology and channel protocols) can have a significant impact on its performance. This motivates the need for fast and accurate exploration tools to evaluate various design alternatives. We have described a trace-based approach that helps evaluate alternative bus protocols and communication architecture topologies. We believe our technique will be helpful in meeting this need in the system-on-chip design environment and have presented experimental results that support claims of efficiency and accuracy.

In future work, we plan to develop methodologies to automatically generate simulation test benches that create worst-case conditions for the communication architecture—for instance, such a test bench could include a set of input stimuli that result in very high contention for access to a shared communication channel. Since our performance-analysis technique is trace-based, use of such test benches will enable exhaustive performance analysis of the system under various boundary conditions. We are currently investigating how to incorporate our performance-analysis tool into an optimization framework to automatically generate an optimal communication architecture for a given system. We are also enhancing the tool to take into account a variety of runtime effects.

ACKNOWLEDGMENT

The authors would like to thank M. Lajolo for his help with the POLIS and PTOLEMY frameworks.

REFERENCES

- [1] S. Dey and S. Bomm, "Performance analysis of a system of communication processes," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1997, pp. 590–597.
- [2] On chip bus attributes specification 1 OCB 1 1.0, on-chip bus DWG, VSI Alliance. [Online]. Available: <http://www.vsi.org/library/specs/summary.htm>

- [3] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1994.
- [4] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Des. Test Comput.*, vol. 10, pp. 64–75, Dec. 1993.
- [5] T. B. Ismail, M. Abid, and M. Jerraya, "COSMOS: A codesign approach for a communicating system," *Proc. IEEE Int. Workshop Hardware/Software Codesign*, pp. 17–24, Sept. 1994.
- [6] A. Kalavade and E. Lee, "A globally critical/locally phase driven algorithm for the constrained hardware software partitioning problem," *Proc. IEEE Int. Workshop Hardware/Software Codesign*, pp. 42–48, Sept. 1994.
- [7] P. H. Chou, R. B. Ortega, and G. B. Borriello, "The CHINOOK hardware/software cosynthesis system," in *Proc. Int. Symp. System Level Synthesis*, Sept. 1995, pp. 22–27.
- [8] B. Lin, "A system design methodology for software/hardware codevelopment of telecommunication network applications," in *Proc. Design Automation Conf.*, June 1996, pp. 672–677.
- [9] J. A. Rowson and A. Sangiovanni-Vincentelli, "Interface based design," in *Proc. Design Automation Conf.*, June 1997, pp. 178–183.
- [10] P. Knudsen and J. Madsen, "Integrating communication protocol selection with partitioning in hardware/software codesign," in *Proc. Int. Symp. System Level Synthesis*, Dec. 1998, pp. 111–116.
- [11] K. Hines and G. Borriello, "Optimizing communication in embedded system cosimulation," in *Proc. Int. Workshop Hardware/Software Codesign*, Mar. 1997, pp. 121–125.
- [12] T. Yen and W. Wolf, "Communication synthesis for distributed embedded system," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1995, pp. 288–294.
- [13] 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*, Sept. 1995, pp. 150–155.
- [14] M. Gasteier and M. Glesner, "Bus-based communication synthesis on system level," *ACM Trans. Des. Autom. Electron. Syst.*, pp. 1–11, Jan. 1999.
- [15] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufman, 1989.
- [16] WARTS: Wisconsin architectural research tools set, Comput. Sci. Dept., Univ. Wisconsin. [Online]. Available: <http://www.cs.wisc.edu/larus/warts.html>
- [17] 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. Int. Workshop Hardware/Software Codesign*, Mar. 1998, pp. 117–121.
- [18] 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 System: The POLIS Approach*. Norwell, MA: Kluwer, 1997.
- [19] J. Buck, S. Ha, E. A. Lee, and D. D. Masserchmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Comput. Simul.*, vol. 4, pp. 155–182, Apr. 1994.



Kanishka Lahiri (S'98) received the B.Tech degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India, in 1998, the M.S. degree in electrical and computer engineering from the University of California, San Diego, in 2000, and is currently working toward the Ph.D. degree in electrical and computer engineering at the same university.

His research interests include design methodologies and architectures for embedded systems, focusing on system-level performance analysis,

power estimation, and architectures for high-performance on-chip communication.

Mr. Lahiri received a Best Paper Award at the 37th Design Automation Conference in 2000.



Anand Raghunathan (S'93–M'97–SM'00) received the B.Tech. degree in electrical and electronics engineering from the Indian Institute of Technology, Madras, India, in 1992, and the M.A. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, in 1994 and 1997, respectively.

He is currently a Research Staff Member at the Computers and Communications Research Laboratories, NEC USA, Princeton, NJ, where he is involved in the research and development of

system-on-chip architectures, high-level design methodologies, and design tools, with emphasis on high-performance and low power design, testing, and design-for-testability. He coauthored *High-level Power Analysis and Optimization* (Norwell, MA: Kluwer, 1998) and holds or has filed for 11 U.S. patents in the areas of system-on-chip architectures, design methodologies, and VLSI CAD. He has presented conference tutorials and on low-power design, and considering testability during high-level design.

Dr. Raghunathan received Best Paper Awards at the 11th IEEE International Conference on VLSI Design (1998) and at the ACM/IEEE Design Automation Conference (1999 and 2000). He received two Best Paper Award nominations at the ACM/IEEE Design Automation Conference (1996 and 1997). He serves on the technical program committees of the ACM/IEEE Design Automation Conference (2000–2001), the IEEE/ACM International Conference on Computer-Aided Design (2000), the IEEE VLSI Test Symposium (1998–2001), the Asia South-Pacific Design Automation Conference (2000), and the International Test Synthesis Workshop (1998–2001). He is the Vice-Chair of the Tutorials and Education Group at the IEEE Computer Society Test Technology Technical Council and an Associate Editor of IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS and IEEE DESIGN AND TEST OF COMPUTERS.



Sujit Dey (S'90–M'91) received the Ph.D. degree in computer science from Duke University, Durham, NC, in 1991.

He is an Associate Professor in the Electrical and Computer Engineering Department at the University of California, San Diego. Prior to joining the University of California, San Diego (UCSD) in 1997, he was a Senior Research Staff Member with the Computers and Communications Research Laboratories, NEC USA, Princeton, NJ. He has presented numerous full-day and embedded tutorials and participated in panels on the topics of hardware–software embedded systems, low-power wireless systems design, and deep submicrometer system-on-chip design and test. He is affiliated with the DARPA/MARCO Gigascale Silicon Research Center and the Center for Wireless Communications at UCSD. He has authored or coauthored one book, 90 journal and conference papers, and ten U.S. patents. His research group at UCSD develops innovative hardware–software architectures and methodologies to harness the full potential of nanometer technologies for future networking and wireless appliances.

Dr. Dey received Best Paper Awards at the Design Automation Conferences in 1994, 1999, and 2000, and the 11th VLSI Design Conference in 1998, and several Best Paper nominations. He has been the General Chair, Program Chair, and member of organizing and program committees of several IEEE conferences and workshops.