Performance Analysis of a System of Communicating Processes

Sujit Dey and Surendra Bommu
C&C Research Laboratories, NEC USA, Inc.
Princeton, NJ 08540

ABSTRACT

Efficient exploration of the system design space necessitates fast and accurate performance estimation as opposed to the computationally prohibitive alternative of exhaustive simulation. This paper addresses the issue of worst-case performance analysis of a system described as a set of concurrent communicating processes. We show that the synchronization overhead associated with inter-process communication can contribute significantly to the overall system performance. Application of existing performance analysis techniques, which target single process descriptions, lead to inaccurate performance estimates as the synchronization overhead is not accounted for. We present PERC, a fast and accurate worst-case performance analysis technique which analyzes inter-process communication, and accounts for synchronization overhead while computing the worst-case performance estimate of a given system implementation. Application of PERC to example systems described as multiple communicating processes shows the ability of the proposed method to accurately estimate the worst-case performance of the system implementation.

I. INTRODUCTION

Increasing complexity, aggressive design requirements, and shorter product cycles are necessitating fast and efficient exploration of the system design space, requiring the capability of analyzing the system performance, power, and other design metrics for several system implementations at high levels of abstraction. This paper introduces a technique to estimate the worst-case performance of a system described as a set of concurrent communicating processes. The performance of a system is determined by both the number of clock cycles required by the system implementation, as well as the clock period of the implementation. A possible way of evaluating the worst case system performance is by exhaustive simulation of the system implementation. However, the approach can be computationally inhibitive, especially in an iterative design framework, where at every iteration, multiple implementation alternatives need to be evaluated. Hence the need for fast static performance analysis and estimation tools.

Several performance analysis and delay estimation techniques have been developed recently. In [1, 2], analysis techniques were developed to compute the average number of clock cycles required by a scheduled implementation of a behavioral description. Markov chain model based metrics for evaluating the performance of schedules was suggested in [3]. Several clock period estimation techniques have also been developed [4, 5, 6, 7, 8] for ASIC implementations. Static performance analysis techniques for software implementation on processors have been proposed in [9, 10].

Many real-life systems, including telecommunication and networking applications, embedded control applications, and multimedia applications among others are described as a set of concurrent communicating processes. The clock period estimation of an implementation is not affected whether the system is described and implemented as a single process or as a set of multiple communicating processes; hence, the clock period estimation techniques described above [4, 5, 6, 7, 8] are applicable to such multi-process systems. However, applying existing performance analysis techniques to estimate number of clock cycles [1, 2, 9] of systems described as concurrent communicating processes will lead to inaccurate results. This is because the above performance analysis techniques target single-process descriptions, and the synchronization overhead associated with inter-process communication, which can contribute significantly to the overall performance, is ignored. Synthesis of a system of communicating processes has been considered in [11] but performance analysis has not been addressed. Optimization of a behavioral description of multiple communicating processes specified in VHDL is suggested in [12]. In [13] the analysis of multi-process systems is addressed using a petri-net model, but the associated data structure could become exponentially large. Consequently, there is a need for a fast and efficient analysis technique which estimates the performance (number of clock cycles) of a system of communicating processes.

This paper introduces PERC, a technique to analyze the communication structure of a set of concurrent communicating processes. Given the system implementation, like the schedule of each process, PERC estimates the worst-case performance of the system. The performance analysis technique relies on computing the time taken per iteration of each loop of the system. Loops which communicate with each other are clustered into sets called communication layers, so that the time taken per iteration of each communicating loop in the layer can be computed including the synchronization overhead. The proposed technique is hierarchical in nature, computing the time taken by the loops in the next level of communication layer using the time taken by loops in the current layer, along with the information on loop bounds. The overall system performance is computed when the top level of communication layer is analyzed. Our analysis approach is also able to identify the critical paths in the the system description, which can be used to improve the system performance by guiding the synthesis tools.

In the next section, we motivate the need for a performance analysis technique which takes into account the synchronization overhead due to the communication of multiple concurrent processes. Section III describes the terminology and definitions used in the subsequent sections, and introduces the synchronization graph used to capture the underlying com-

0-89791-993-9/97 $10.00

munication structure of a system. Section IV introduces the concept of communication layers, while Section V discusses the procedure of identifying the communication layers. Section VI gives a method of analyzing all the loops of a single communication layer and computing the time taken by each loop in the layer. Section VII describes the technique to estimate the overall system performance. Results of application of the proposed performance analysis tool PERC to example systems of concurrently communicating processes are provided in Section VIII.

II. Motivation

In this section, we illustrate an example system of concurrent communicating processes. Using the example system, we motivate the need for a global performance analysis technique which analyzes and includes the synchronization overhead while computing the system performance.

Figure 1 shows parts of the control-flow graph (CFG) of an ethernet controller description. The ethernet protocol involves sending data packets over the network. If a particular transmission of a packet is not successful (a collision detected over the network), the packet has to be re-transmitted after a specified amount of time. The protocol also ensures that a minimum amount of time has elapsed between successive transmissions of the packet. The overall communication pattern of the ethernet controller is shown in Figure 1.

Communication between processes can be achieved using various protocols, depending upon the language used to describe the system. Depending on the semantics allowed by a particular language the communication between concurrently executing entities could be of several types, but the underlying concept involves one entity sending an event to another concurrent entity which would be waiting for the event. The systems considered in this paper are assumed to be communicating using event and wait statements as shown in the *ethernet controller* example in Figure 1. An assignment to a global signal generates an event which is received by a process waiting on the event. The global signals are divided into *synchronizing signals* and *data transfer signals*. The synchronizing signals are used to generate events and synchronize among the processes, while the data transfer signals are used to exchange information between processes. In the example, shown in Figure-1, the signal *XmitBegin* is a synchronizing signal, whereas the signal *NewCollision* represents a data transfer signal as this signal does not play a role in the synchronization between processes.

To investigate the issue of performance analysis of a system described as a set of concurrent communicating processes, we generated a scheduled implementation of the ethernet controller using a scheduling tool [1]. We then considered the problem of computing the worst-case number of clock cycles needed by the scheduled implementation. The result obtained by simulating the scheduled description using some test cases is shown in Table 2, which is 767 clock cycles. Next, we used an existing performance analysis technique [1] to compute the number of clock cycles taken by each process. Note that since the existing technique(s) can be applied only to a single process at a time, the communication between the processes, and hence the time spent in synchronization (wait statements), is ignored. The result is shown in Table 2. Processes 1, 2, and 3 are estimated to take 175, 48, and 291 respectively. Hence, the worst case performance of the system, which is the maximum of the number of clock cycles needed to execute each process, is 291 clock cycles. The large percentage difference with the simulation result, -62%, represents a significant under-estimate of the worst-case performance of the system. The large error can be attributed to ignoring the synchronization overhead among the processes.

On the other hand, if the performance estimation technique can perform a global analysis of all the processes and their communication, and the time taken in process synchronization is accounted for while estimating the performance of each process, the system performance estimate is much more accurate. The result of applying the performance estimation technique PERC discussed in this paper is also shown in Table 2. The performance of processes 1, 2, and 3 is now estimated to be 902, 334, and 334 clock cycles respectively, with a total system performance to be 902 clock cycles. The PERC estimate has a significantly smaller difference (18%) with the result obtained by simulation. Also, as expected, the PERC estimate is more conservative than that obtained by simulation, which is not exhaustive and cannot always identify the worst-case behavior of a system.

Note the significant difference in clock cycle estimates between PERC and the technique which analyzes each process individually. For example, PERC estimates process 1 to take 902 clock cycles, as opposed to the estimate of 175 clock cycles obtained when time spent in wait statements is ignored. The difference in the estimates is due to the significant time the processes spend in waiting for synchronizing events, and demonstrates the critical need for considering synchronization overhead during performance analysis.

In the next few sections, we describe the proposed performance analysis technique which estimates the worst-case performance taking into account the synchronization overhead.

III. Characteristics of Synchronizing Processes

In this section, we introduce some terminology and definitions used in the paper. We also discuss the assumptions made regarding the allowed communication between processes.

An event or wait statement is termed as a synchronizing statement. A loop l_i is said to contain a synchronizing statement if the statement is present in the loop and is not present in any loop nested by l_i. A loop containing a synchronizing statement will be referred to as a *synchronizing loop*. Since individual processes could iterate more than once, each process itself is considered to be a loop. Consider the description of the ethernet controller shown in Figure 1. The loop representing the process *BitTransmitter* is a synchronizing loop as it contains synchronizing statements like $XmitEnd \leftarrow 1$.

We next introduce the *Synchronization Graph* which captures the communication/synchronization among the system processes. The synchronization graph is a directed graph with vertex set V and edge set E. The set of vertices V consists of the the following types of node: *Loop Nodes* corresponding to the beginning and the end of each loop l in the system description, where loop l is either a synchronizing loop or nests a synchronizing loop; *Event nodes* corresponding to an event statement in the behavioral specification, and *Wait nodes* corresponding to a wait statement in the behavioral specification.

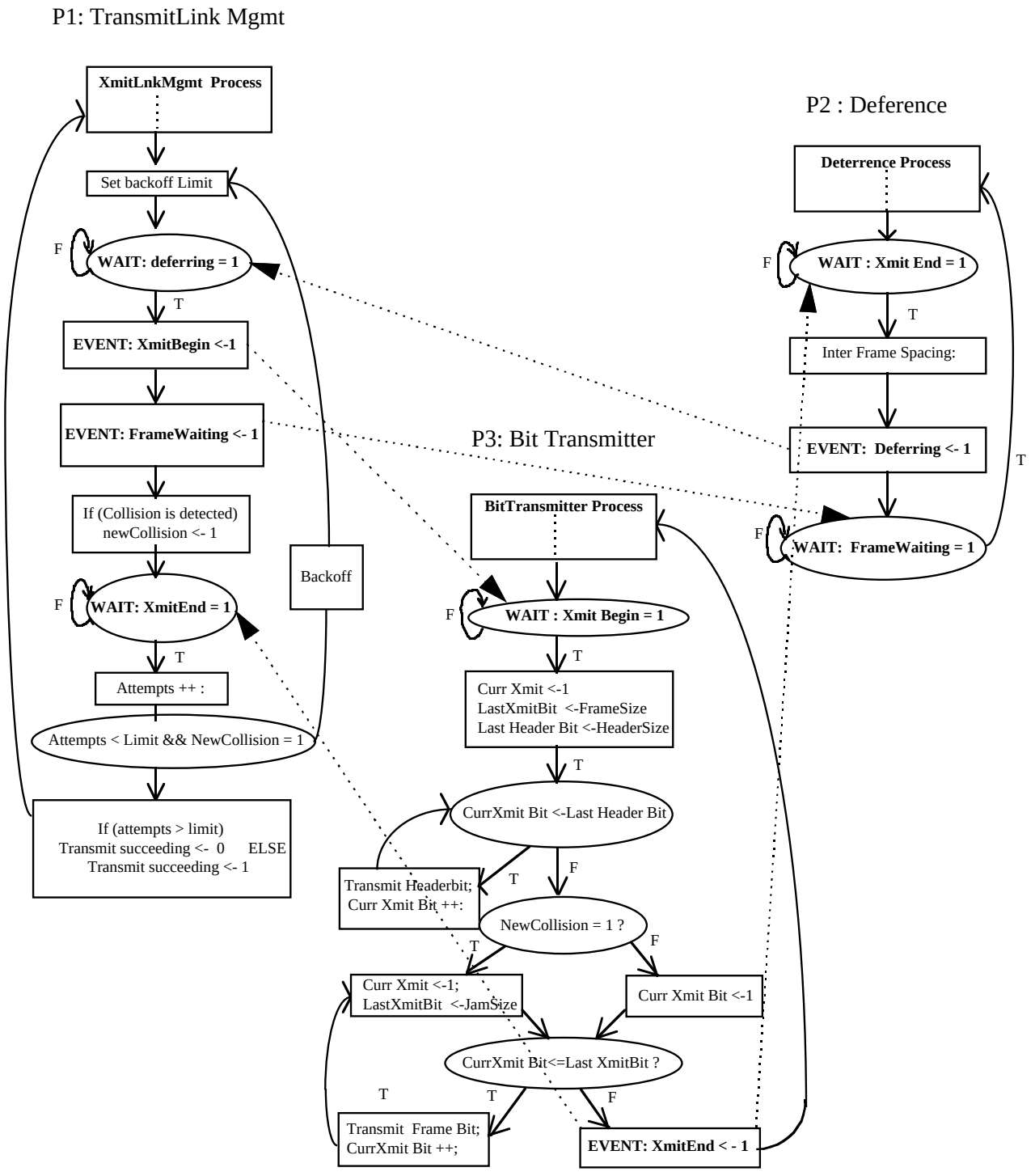

Figure 1: Partial description of an Ethernet Controller, showing inter-process communication

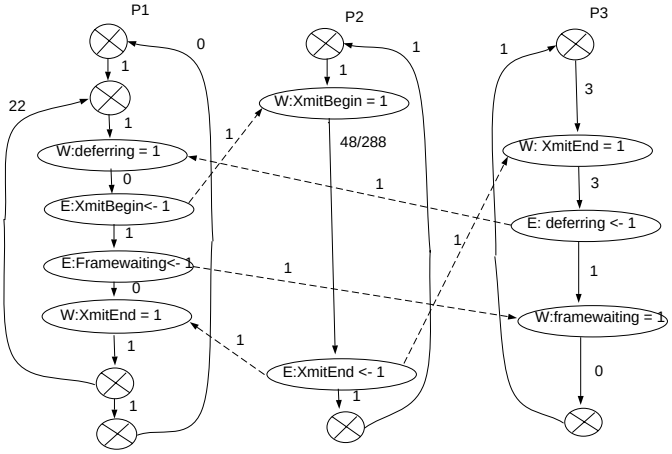

Figure 2: Synchronization Graph of the Ethernet Controller

There exists a process edge between two nodes if they belong to the same process and there is a direct path connecting the corresponding behavioral statements in the system specification, not containing any statement that can be represented as a node in the synchronization graph. The weight of a process edge represents the worst case time needed by the system implementation (in this paper, the schedule) to execute the statements in the corresponding path. There exists a synchronizing edge between an event node (v_1) and wait node (v_2), if v_1 and v_2 do not belong to the same process, and the event generated by v_1 is meant to be received by wait node v_2. The weight associated with a synchronizing edge represents the communication overhead involved in sending the event, and depends on the system implementation.

Figure 2 shows the synchronization graph corresponding to the ethernet controller description partially shown in Figure 1. For example, in the *Deference* process of Figure 1, there is a wait node in the synchronization graph corresponding to the statement *WAIT: XmitEnd = 1*, and an event node in the synchronization graph corresponding to the statement *Event:Deferring* $\leftarrow$ 1. A process edge exists between the above two nodes, representing the block of code *InterFrameSpacing* shown in Figure 1. The weight of the process edge is 3, since it takes 3 clock cycles in the worst case to execute the statements corresponding to the process edge. A synchronizing edge exists between the node representing *Event: Deferring* $\leftarrow$ 1 (process Deference) and the wait statement *WAIT: Deferring* $\leftarrow$ 1 of process *TransmitLinkMgmt*. In this example, communication between the processes is assumed to go through registers, requiring a single clock cycle; hence, each synchronizing edge has been assigned a weight of 1. In general, the weight assigned to each synchronizing edge will depend upon the mode of communication implemented.

Each synchronizing loop, l_i, is associated with a quadruple $(N(l_i), W(l_i), E(l_i), P(l_i))$, where *N($l_i$)* represents the set of synchronizing loops nested by the synchronizing loops l_i, *E(l_i)* represents the set of synchronizing loops which wait on an event generated inside l_i, *W(l_i)* represents the set of synchronizing loops which generate an event that l_i could be waiting on, and P(l_i) represents the process containing the loop l_i.

We shall now discuss some of the assumptions made regarding the communication between concurrent processes. As mentioned before, the system communicates using event and wait statments. An event is generated by setting a synchronizing variable to one, and the synchronizing variable is reset by the receiving process immediately after receiving the event and coming out of the wait statement. The following assumptions are made regarding the generation/consumption of events: (1) The rate of generation of an event is less than or equal to the rate of consumption of the corresponding event. Relaxing this assumption would mean that buffers will be needed to store the outstanding events. In this paper we will not consider system implementations which use buffers to store the events which have not yet been consumed, and (2) A synchronizing signal cannot be assigned to or waited on more than once in a loop. This assumption can be enforced as a semantic restriction; while not being too restrictive on the designer, this constraint facilitates the analysis of the system.

In the next section, we propose a method to analyze and take into account the effect of synchronization overhead in system performance.

IV. Considering Synchronization Overhead in Performance Analysis

To compute the performance of a system, the time taken by the loops of the system description have to be computed. In a system of communicating processes, the time taken by a synchronizing loop containing a wait statement depends not only on the time to execute the statements in the loop, but also on the *synchronization overhead*, which is the time taken by the wait statements in waiting for events generated by other communicating loops. To enable our performance analysis to include the synchronization overhead, we introduce the concept of a *communication layer*, which is a set of synchronizing loops communicating with each other simultaneously. Considering all the loops in a communication layer together during performance analysis is sufficient to account for the synchronization overhead. Next, we describe the notion of communication layers.

A loop l_i is said to *communicate* with another loop l_j if either l_i has a wait node waiting for an event generated in l_j, or vice versa. That is, $l_i \subset (E(l_j) \cup W(l_j))$. A loop is said to be *alive* in a given clock cycle, if any node (statement) in the loop is executed in that cycle. If two loops are simultaneously alive in a given clock cycle, they are said to be executing *concurrently*. A **communication layer** is defined as a maximal set of synchronizing loops such that

C1 each loop of the set communicates with at least one other loop of the set, and

C2 all the loops of the set should execute concurrently.

Consider the synchronization graph shown in Figure 3, which corresponds to the description of a blackjack game controller, containing three communicating processes. Consider the set of loops $L = \{l_1, l_2, l_3, l_6, l_7\}$ in Figure 3. Each loop of the set communicates with at least one other loop of the set. For instance, l_7 communicates with l_6, and each of l_1, l_2, l_3 communicates with l_6. Consequently, the set L satisfies condition C1 of the definition of a communication layer. Loops l_1, l_2, l_3 do not execute concurrently (violates condition C2), and hence do not belong to the same layer. The set of loops which are concurrent are $L_1 = \{l_1, l_6, l_7\}$,

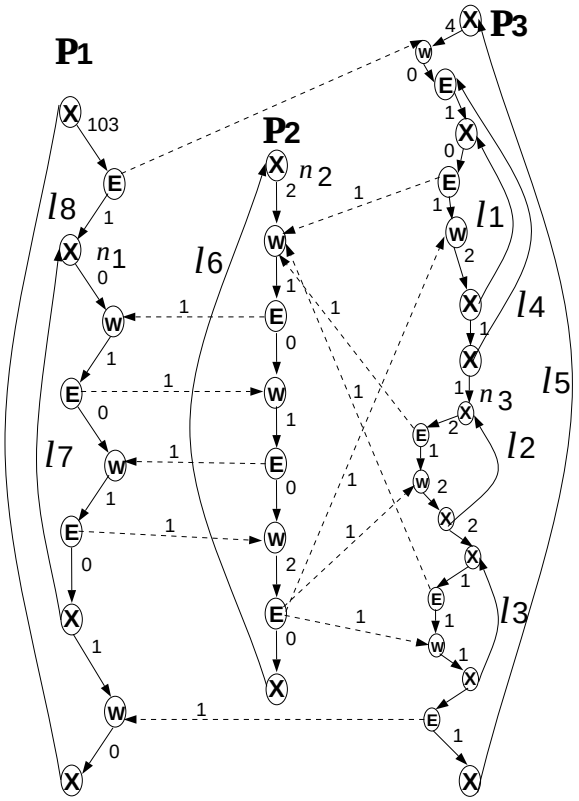

Figure 3: Synchronization Graph of Blackjack Game Controller

$L_2 = \{l_2, l_6, l_7\}$ and $L_3 = \{l_3, l_6, l_7\}$. Since the loops in the above sets also satisfy condition C1, they form three communication layers. The set $L_4 = \{l_8, l_5\}$ forms a fourth communication layer.

A layer L_1 is said to nest a layer L_2 if a loop in L_1 nests a loop belonging to L_2. That is, $\exists$ a loop $l \in L_1$ such that $N(l) \cap L_2 \neq nil$. Let $N(L)$ be the set of layers nested by L. The *nesting level* of a layer L, $Level(L)$, is defined as:

$$Level(L) = \begin{cases} 1 & \text{if } N(L) = nil \\ \max_{L_i \in N(L)} Level(L_i) + 1 & \text{otherwise} \end{cases}$$

Consider the layers formed above for the synchronization graph of Figure 3. None of the loops in L_1, l_2 and L_3 nest other loops and hence the nesting level of L_1, L_2 and L_3 is 1. In the layer $L_4 = \{l_8, l_5\}$, loop l_5 nests the loops l_1, l_2, and l_3. Since $l_1 \in L_1, l_2 \in L_2$, and $l_3 \in L_3$, we have the nesting level of L_4 = 1 + Max(nesting levels(L_1, L_2, L_3)) = 2.

Our performance analysis method analyzes each communication layer in an increasing order of their nesting level. For each layer, the time taken by each loop belonging to the layer is computed as explained in Section VI. After a communication layer L is analyzed, the loops in L are collapsed, and the layer nesting L is analyzed using the time taken by the collapsed loops. This process is continued till all the layers have been analyzed; the analysis of the final layer gives the system performance. The overall system analysis method is explained in Section VII. In the next section, the procedure for identifying communication layers along with their nesting levels is presented.

V. Identifying Communication Layers

The formation of communication layers needs information about the concurrent execution of two synchronizing loops, which depends on the pattern of communication and the characteristics of the implementation of the system. In the synchronization graph shown in Figure 3, loops l_6 and l_7 execute concurrently, but loops l_1, l_2, l_3 do not. In the subsequent discussions, the concurrency information for communicating loops is assumed to be available.

A. Creating Layers

In this section we shall describe a procedure for identifying the layers from a set of loops S. First, the set of loops s is partitioned into classes. A class is a maximal subset of communicating loops, A, of S satisfying the following conditions: (1) all the loops of subset A should communicate with at least one other loop of A, and (2) the loops of A do not communicate with any loops outside A.

The members of a class, clearly, satisfy condition C1 of a communication layer. In Figure 3, the set $\{l_1, l_2, l_3, l_6, l_7\}$ forms a class. The concurrency information between synchronizing loops is then used to further divide each class into layers. The algorithm given below partitions a class into communication layers and returns a list of communication layers formed.

Algorithm 1 $ClassToLayers(Class\ C)$

```
/* C represents set of loops belonging to the same class */
/* If class is already a layer return*/
1.    if(∀l_i, l_j ∈ C, l_i and l_j are concurrent)
2.        return C;
/* If class is not a layer do the following */
3.    if(∃l_i, l_j ∈ C | l_i l_j are not concurrent){
4.        C^o = C − {l_i, l_j};
5.        C_1 = C^o ∪ {l_i}; C_2 = C^o ∪ {l_j};
6.        return ClassToLayers(C_1)
                  ∪ ClassToLayers(C_2);
      }
```

The above recursive algorithm returns a list of layers formed from the class C, given as input to the algorithm. We shall illustrate the execution of Algorithm 1 using the set $C = \{l_1, l_2, l_3, l_6, l_7\}$ of the synchronization graph in Figure 3. In the first iteration of Algorithm 1, the set C is divided into the sets $C_1 = \{l_6, l_7, l_2, l_3\}$ and $C_2 = \{l_6, l_7, l_1, l_3\}$, since l_1 and l_2 do not execute concurrently. A recursive call is made on each of the sets C_1 and C_2. The loops l_2, l_3 of C_1 and l_1, l_3 of C_2 do not execute concurrently, hence the classes C_1 and C_2 are further divided into the following: $C_{11} = C_{21} = \{l_6, l_7, l_3\}$, $C_{12} = \{l_6, l_7, l_1\}$ and $C_{22} = \{l_6, l_7, l_2\}$; each of the above classes are layers since all the loops in the class run concurrently. Hence the layers formed are: $L_1 = C_{12} = \{l_1, l_6, l_7\}$, $L_2 = C_{22} = \{l_2, l_6, l_7\}$ and $L_3 = C_{11} = C_{22} = \{l_3, l_6, l_7\}$.

B. Identifying Layers and their nesting levels

In this section, the overall procedure used to identify all the communication layers is given, first the pseudo-code, followed by a brief description.

Algorithm 2 $Identify\_Layers(U)$

```
level = 1;
While(U ≠ nil){
    S = {l | l does not nest any loop in U};
    Layer[level] = Create_Layers(S);
    /*Each element of the array Layer represents */
    /*all the layers of the same level */
    U = U − {l | l ∈ Layer[level]};
    level++;
}
```

The input to the above algorithm is the set of all synchronizing loops of the system. The function *Create_Layers* identifies layers from a set of loops S by first dividing S into classes, and then using the function *ClasstoLayers* (section IV) to compute the layers of each class. At each iteration i of the $while$ loop, layers of level i are formed. All the loops which are partitioned into layers are removed from consideration for partitioning in the remaining iterations; the procedure is continued till all the loops are partitioned. Consider the blackjack example shown in Figure 3. The set U is initialized to all the loops of the system. In the first iteration the set S contains $\{ l_1, l_2, l_3, l_6, l_7\}$, and the corresponding layers formed are given by the sets L_1, L_2, L_3 as described in section A. Hence the nesting level of layers L_1, L_2, L_3 is 1. In the second and final iteration, set $S = \{l_8, l_9\}$, yielding the fourth layer $L_4 = \{l_8, l_9\}$, with a nesting level of 2.

VI. Analyzing a Single Layer of Communication

In this section, we discuss a technique to analyze a given communication layer. We first show that the time taken per iteration of all the loops in a layer will be the same. Subsequently, we describe a technique to analyze the layer, and compute the time taken by the loops in the layer.

Lemma 1: The time taken per iteration of all loops in a communication layer are equal.

Proof: Consider loop l_i communicating with another loop l_j. Say $l_j \subset E(l_i)$, that is, l_i generates an event to be received by a wait node in l_j. Also, say the time taken per iteration of loop l_i is T_{l_i}, and that of loop l_j is T_{l_j}. Since l_j has to wait on every iteration for the generation of the event by l_i, we have: (1) $T_{l_j} \geq T_{l_i}$. Since the rate of generation of events is assumed to be less than their consumption, and with the restriction that each event is generated or received only once in every iteration of a loop (this is one of the constraints enforced on the communication pattern of the system, as described in Section III), we have: (2) $\frac{1}{T_{l_i}} \leq \frac{1}{T_{l_j}}$. From equation (2), we have: (3) $T_{l_i} \geq T_{l_j}$.

From equations (1) and (3), we get $T_{l_i} = T_{l_j}$. By definition, all the loops in a layer communicate with each other, hence the time taken per iteration for all the loops in a layer is the same. □

Next, we describe a method to compute the time taken by each loop in a layer. Associated with each layer is a *layer graph*, which is a cyclic subgraph of the synchronization graph consisting of the nodes and edges of the loops belonging to the layer. There is a start node and end node for each loop in the layer graph. For example, in the synchronization graph of Figure 3, n_1, n_2 and n_3 represent the start nodes of loops l_6, l_7 and l_2 respectively of the layer graph of layer L_2.

Our method involves computing the arrival times of the nodes in the layer graph. The arrival time of a node v, a(v), is computed in terms of the arrival times of its fanin nodes as:

$$a(v) = \max_{v_i \in Fanin(v)} a(v_i) + Edge\_Weight(v_1 \rightarrow v). \quad (1)$$

Clearly, the arrival times of fanin nodes need to be known before the arrival time of a node can be determined, which is a problem in a cyclic graph like the layer graph. We account for the cyclic nature of the layer graph as follows. Initially, the arrival times of the start nodes are set to 0. Also, the synchronizing variables corresponding to some event nodes may be initialized in the behavioral description. The arrival times of such event nodes will be set to 0 in the layer graph.

Beginning with the loop start nodes, a breadth first traversal of the layer graph produces the arrival times of all the loop end nodes. The time taken by a loop is given by the difference between the arrival times of the end and start nodes of the loop. At the end of the first iteration, the time taken by the various loops in the layer may be different. Hence, the timing analysis procedure is iterated, recomputing the arrival times of the nodes from the start nodes to the end nodes, till the time taken by all the loops are equal. Note that from the second iteration, the arrival times of each start node is computed using the arrival times of the end nodes in the previous iteration, and the weight of the back edges.

VII. Analysis of Multi-Layer Communication Systems

In this section, we shall discuss the overall strategy for the worst case performance analysis of a system of concurrent communicating processes. At first, the synchronization graph is partitioned into communication layers, and their nesting levels identified. The time taken per iteration of loops in a layer L_j depends on the time taken by the loops in layers nested by L_j. Hence, the layers are analyzed in an increasing order of their nesting levels, so that all the layers nested by layer L_j are analyzed before layer L_j is analyzed. Consider the layers of the synchronization graph of Figure 3 which are derived in section V. The layers L_1, L_2, L_3 are of level 1, and hence analyzed first. Subsequently, the layer L_4, of level 2, is analyzed.

Before analyzing a communication layer of level i, $i > 1$, all the layers of level $i-1$ are analyzed and collapsed. Collapsing a layer effectively removes the corresponding loops from the synchronization graph, updating the graph accordingly to reflect the time taken by the collapsed layer. The time taken by a layer is the time taken by the loops in the layer, plus the time taken to execute other statements in the layer not included in the loops. The time taken by each loop in a layer is the product of the time taken per iteration of the loop and the corresponding loop bound. As discussed in Section VI, the time taken per iteration of all the loops in a layer are identical, and are simultaneously computed. We next briefly discuss the issue of loop bounds. Since all the synchronizing loops of a layer run concurrently and communicate with each other, the loop bounds for each of the loops in the layer are assumed to be equal. The loops which belong to more than one layer can have more than one loop bound, depending on the particular layer under consideration. The loop bounds of each layer is extracted from the behavioral description with some feedback from the user.

Collapsing a layer removes all the nodes and edges of the

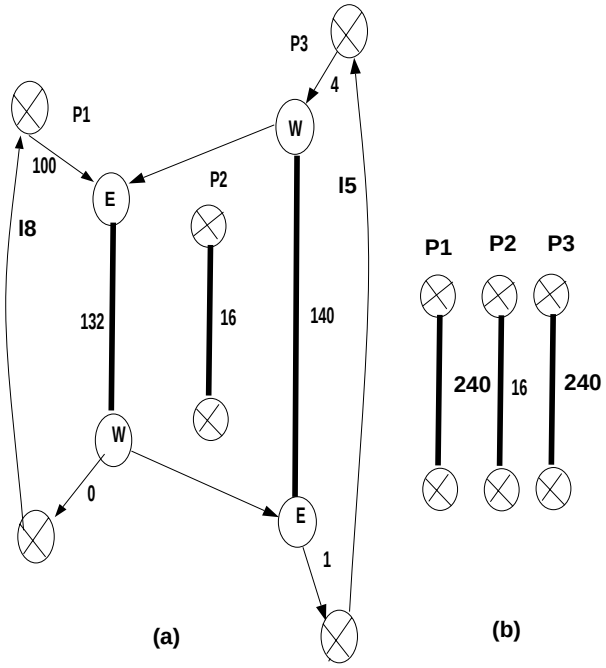

Figure 4: (a) Synchronization graph after collapsing loops in level one layers, (b) Final Synchronization graph after collapsing loops in level two layer

loops of the layer being collapsed, replacing each loop with an edge representing the time taken by the loop. If a loop belongs to more than one layer, it is removed from the synchronization graph only after all the corresponding layers are collapsed and the edge weight of the added edge is equal to: (1) The sum of the delays associated with each of the layers, if the loop is not a process loop. This is to reflect the fact that the time taken per iteration of a process depends on the overall time taken by each of the loops nested by the process loop. (2) The maximum of the delays, if the loop corresponds to a process loop. This is because we just need in the worst case the time taken per iteration of each process to determine the system performance. Consider the blackjack example shown in Figure 3. Loop l_7 is present in three layers, L_1, L_2, L_3 (section V), and it does not correspond to a process loop. Hence, l_7 is replaced by an edge of weight equal to the sum of the delays of layers L_1, L_2, and L_3. On the other hand, loop l_6 belongs to more than one layer (L_1, L_2, L_3) but represents a process loop; hence, l_6 is replaced by an edge with weight equal to the maximum among the layers L_1, L_2, L_3. The synchronization graph for the *blackjack* example after collapsing layers L_1, L_2 and L_3 is given in Figure 4(a), with the new edge weights shown.

Since the collapsing of a layer involves the removal of the associated synchronizing edges, after all the layers of communication are analyzed and collapsed, the synchronization graph will not contain any synchronizing edges. The final graph would be a collection of independent components, each component being an edge representing a process of the system. Figure 4(b) shows the final synchronization graph after layer L_4 is analyzed and collapsed. The component with maximum edge weight represents the process which requires the maximum number of clock cycles to complete, and hence is the system performance. In the *blackjack* example, the worst case system performance is 240 clock cycles.

The overall performance analysis procedure can be summarized as follows: (1) Create synchronization graph. (2) Identify communication layers and their nesting levels. (3) For each layer in increasing nesting level: (a) Compute time taken by the layer, (b) Collapse layer and update synchronization graph. (4) System performance = weight of maximum weighted edge/component of the final synchronization graph.

VIII. Experimental Results

A prototype of the proposed performance analysis technique, PERC, has been developed. In this section, we shall present the results of applying PERC to three systems, an ethernet controller, a blackjack game controller, and a DMA controller, the first two examples having been discussed in this paper. Each system has been described as a set of concurrent communicating processes. Table 1 reports some characteristics of the behavioral description and the corresponding synchronization graph of the three systems. The number of processes and the lines of behavioral code of the systems are given under the system description column. The number of synchronizing nodes, synchronizing edges, and communication layers of the corresponding synchronization graph are given in the next few columns. For example, the description of the ethernet controller consists of 3 processes and 323 lines of code, while the corresponding synchronization graph contains 9 synchronizing nodes, 5 synchronizing edges, and one communication layer.

Table 1: Characteristics of the systems

	System Description		Synch. Graph		
	Process	Lines of code	Synch. nodes	Synch. edges	Comm. Layers
Ethernet	3	323	9	5	1
Blackjack	3	399	18	12	4
DMA	3	241	20	10	2

For each system description, we generated a scheduled implementation using a scheduling tool [1]. Next, we obtained the worst-case performance (number of clock cycles) taken by the scheduled implementation using (1) simulation with functional test cases, (2) static performance analysis without considering the inter-process communication (considering each process separately), and (3) static performance analysis using PERC, which includes synchronization overhead. For the analysis part, the edge weights of the synchronization graph are computed using the scheduled implementations of the individual processes. Table 2 reports the worst-case system performance as computed by the three above mentioned approaches, in columns `Simulation` (S), `Performance Estimate without synchronization overhead` (E_1), and `Performance Estimate using PERC` (E_2), respectively. While the performance of the complete system is reported in the column `System`, the performance estimates of individual processes are reported in the columns `P1`, `P2`, and `P3` respectively. Finally, for the results obtained using the analysis methods, the percentage difference between the performance numbers obtained by the analysis method and simulation, $(E_1 - S)/S * 100$, and $(E_2 - S)/S * 100$, are shown in the two columns Δ_1 and Δ_2

Table 2: Performance analysis results

	Simulation (S)	Performance Estimate without synch overhead (E_1)					Performance Estimate by PERC (E_2)				
		P_1	P_2	P_3	System	Δ_1%	P_1	P_2	P_3	System	Δ_2%
Ethernet	767	175	48	291	291	-62	902	334	334	902	+18
Blackjack	205	121	7	41	121	-40	240	16	240	240	+17
DMA	67	29	42	13	42	-37	74	50	23	74	+10

$\Delta_1 = (E_1 - S)/S * 100$; $\Delta_2 = (E_2 - S)/S * 100$

respectively. Note that the performance estimates obtained by the analysis method which does not consider inter-process communication grossly under-estimates the worst-case performance, as is reflected by the percentage difference with simulation Δ_1. Such under-estimation of performance may not be acceptable for certain systems, like real-time systems. On the other hand, the performance estimates obtained by PERC have significantly smaller percentage difference with the simulation results, as shown by the column Δ_2. Also, the estimates obtained by PERC are always more conservative than the simulation results; this is expected since simulation, unless exhaustive, can only produce a lower bound of the worst case performance.

For example, for the ethernet controller, simulation produced a worst-case performance of 767 clock cycles. Using existing performance analysis techniques without considering the synchronization overhead, the performance estimates of the individual processes P1, P2, and P3 are 175, 48, and 291 clock cycles respectively, giving a total system performance of 291 clock cycles. The large percentage difference with the simulation result, -62%, represents a significant under-estimate of the worst-case performance of the system. The large error can be attributed to ignoring the synchronization overhead of inter-process communication. Using PERC, the performance estimates obtained for processes P1, P2, and P3 are 902, 334, and 334 clock cycles respectively, with a total system performance of 902 clock cycles. The PERC estimate has a significantly smaller difference (18%) with the result obtained by simulation. Also, as expected, the PERC estimate is more conservative than that obtained by simulation, which is not exhaustive and cannot always identify the worst-case behavior of a system.

The difference in the estimates is due to the significant time the processes spend in waiting for synchronizing events, and demonstrate the critical need for considering synchronization overhead during performance analysis. The results show that the proposed technique, by considering synchronizing overhead, can obtain fairly accurate performance estimates, unlike existing performance analysis techniques, which do not consider inter-process communication.

IX. Conclusion

This paper introduced a static performance analysis technique for estimating the worst-case performance of a system of concurrently communicating processes. The need for analyzing and estimating the synchronization overhead incurred by the communicating processes, to obtain accurate estimate of the system performance, has been demonstrated. The proposed technique has been applied to analyze systems implemented in hardware. However, as long as the worst case time taken between any two synchronization points of a system can be computed, the analysis technique can be used independent of the actual implementation of individual processes. We are currently studying the possibility of extending *PERC* to analyze systems implemented as mixed software and hardware. Also, we are exploring ways of using the information regarding the critical parts of the system in a synthesis framework to improve the system performance.

ACKNOWLEDGEMENT The authors would like to thank S. Biswas and A. Raghunathan for helpful discussions, and T. Misawa for help with the designs.

References

[1] S. Bhattacharya, S. Dey, and F. Brglez, "Performance Analysis and Optimization of Schedules for Conditional and Loop-Intensive Specifications," in *Proc. Design Automation Conference*, pp. 491–496, June 1994.

[2] M. Rahmouni and A. Jerraya, "Formulation and Evaluation Of Scheduling Techniques For Control Flow Graphs," in *Proc. of the European Design Automation Conference*, Sept. 1995.

[3] F. Brewer and D. Gajski, " Chippe: A System for Constraint Driven Behavioral Synthesis ," *IEEE Transactions on Computer Aided Design*, vol. Vol. 9, pp. 681–695, July 1990.

[4] A. Kuehlmann and R. A. Bergamaschi, " Timing Analysis in High-Level Synthesis," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, August 1992.

[5] S. Narayan and D. D. Gajski, "System Clock Estimation based on Clock Slack Minimization," in *Proc. of the European Design Automation Conference*, 1992.

[6] C. Ramachandran and F. J. Kurdahi, "Combined Topological and Functionality Based Delay Estimation Using a Layout-Driven Approach for High Level Applications," *IEEE Transactions on Computer-Aided Design*, vol. 13, pp. 1450 – 1460, Dec. 1994.

[7] S. Bhattacharya, S. Dey, and F. Brglez, "Fast True Delay Estimation During High Level Synthesis," *IEEE Trans. on Computer-Aided Design*, vol. 15, pp. 1088 – 1105, Sept 1996.

[8] C. Safinia, R. Leveugle, and G. Saucier, "Taking Advantage of High Level Functional Information to Refine Timing Analysis and Timing Modeling," in *Proceedings of the European Design Automation Conference*, 1994.

[9] Y.-T. S. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," in *IEEE/ACM International Conference on Computer Aided Design.*, pp. 380–7, November 1995.

[10] K. Suzuki and A. Sangiovanni-Vincentelli, "Efficient software performance estimation methods for hardware/software codesign," in *33rd Design Automation Conf.*, pp. 605–610, 1996.

[11] W. Wolfe, A. Takach, C.-Y. Huang, and R. Manno, "The princeton university behavioral synthesis system," in *Proc. 29th Design Automation Conference*, pp. 182–7, 1992.

[12] F. Martinolle, "Fusion of vhdl processes," tech. rep., Center For Reliable Computing, Computer Systems Laboratory, Department of Electrical and Computer Engineering, Stanford University, 1991.

[13] J. Miller and H. Kramer, "Analysis of multi-process specifications with a petri-net model," in *Proceedings EURO-DAC'93. European Design Automation Conference with EURO-VHDL'93*, pp. 474–9, 1993.