Efficient Performance Analysis of Asynchronous Systems Based on Periodicity

Peggy B. McGee Steven M. Nowick *
Department of Computer Science
Columbia University
New York, NY 10027
{pmcgee,nowick}@cs.columbia.edu

E. G. Coffman Jr.
Department of Electrical Engineering
Columbia University
New York, NY 10027
egc@ee.columbia.edu

ABSTRACT

This paper presents an efficient method for the performance analysis and optimization of asynchronous systems. An asynchronous system is modeled as a marked graph with probabilistic delay distributions. We show that these systems exhibit inherent periodic behaviors. Based on this property, we derive an algorithm to construct the state space of the system through composition and capture the time evolution of the states into a periodic Markov chain. The system is solved for important performance metrics such as the distribution of input arrival time at a component, which is useful for subsequent system optimization, as well as relative component utilization, system latency and throughput. We also present a tool to demonstrate the feasibility of this method. Initial experimental results are promising, showing over three orders of magnitude improvement in runtime and nearly two orders of magnitude decrease in the size of the state space over previously published results. While the focus of this paper is on asynchronous digital systems, our technique can be applied to other concurrent systems that exhibit global asynchronous behavior, such as GALS and embedded systems.

Categories and Subject Descriptors:
C.4 [Computer Systems]:Performance of Systems
General Terms: Algorithms, Performance
Keywords: asynchronous, performance, periodic, marked graphs, Petri nets

1. INTRODUCTION

One potential advantage of asynchronous systems over their clocked counterparts, in certain applications, is better average-case performance [1, 19, 25]. However, the performance analysis of asynchronous systems remains a challenge. Without an effective performance analysis method, one cannot easily take advantage of the properties of asynchronous systems to achieve optimal performance.

There are several major technical difficulties involved in the performance analysis of asynchronous systems. First, unlike clocked systems where clock boundaries form natural partitions for logic between stages to be analyzed individually, an asynchronous system is inherently nonlinear, meaning there is no easy way to partition the system into independent subsystems. The system has to be analyzed as a whole. This has often led to unmanageable state space problems. Second, as the system is event-driven, arbitrary arrival time of inputs and variations in data-dependent delays in individual components can have significant impact on the overall performance of the system, and must therefore be taken into account during performance analysis. As pointed out in [17], taking the statistical average of the processing time of individual components is often inadequate in determining the average performance of the overall system; the so-called "variance" of processing times must also be considered. Finally, there is no clear consensus in the asynchronous design community on what performance metrics are useful for characterizing the performance of a system and for identifying bottlenecks for optimization.

This paper addresses these issues by providing an efficient and general method for analyzing the asymptotic performance of asynchronous systems. In our approach, an asynchronous system is modeled as a marked graph, a subclass of Petri nets that captures concurrency and data-dependent relationships between interacting components in decision-free systems [7].[1] The variations in input arrival time and component delays are captured in a probabilistic delay model. The probability distribution of input arrival at a component can provide indication of system bottlenecks. Component utilization, as well as system latency and throughput, can be derived as measures of system performance.

This paper focuses on the subclass of asynchronous systems that can be modeled by strongly-connected marked graphs, namely, decision-free systems. A sketch of how to extend our approach to systems with choice is presented in Section 7. The subclass of decision-free systems, while limited, has been shown to be capable of modeling many interesting concurrent systems [11]. More importantly, it forms a foundation to develop theories and algorithms for analyzing more complex systems, which we will consider in future work. We also focus on modeling at the system architecture level; we do not currently consider circuit-level modeling issues such as delay variations due to process variations.

There are other formalisms that are used in practice to model deterministic concurrent systems besides marked graphs, such as synchronous dataflow languages [15] to model DSP's, and Kahn-type models [13]. If one assumes bounded buffering, these models can be used as input languages for specifying a system, which can then be translated to marked graphs, and our method can be applied.

Previous work that considers probabilistic delay models in the performance analysis of asynchronous systems falls into three classes. The first uses a queuing network model and gives bounds on utilization. Greenstreet and Steiglitz [10] analyzed the performance of self-timed pipelines with exponential processing times. Pang

*This work was supported by NSF ITR Award No. NSF-CCR-0086036.

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

[1] These systems are also referred to as "deterministic" concurrent systems, or concurrent systems without conditional behavior (or without "choice"), in the literature.

and Greenstreet [17] extended the analysis to the performance of self-timed meshes. This method, however, can only be applied to systems that exhibit regular architectures. In contrast, the current paper handles arbitrary architectures.

A second class of methods is based on simulation. Xie and Beerel [24] gave bounds on average time separations of events in an asynchronous system using a simulation-based method. While simulation-based methods are indispensable in other parts of the design flow, they may be too expensive to run in the system-level optimization loop, as often many simulation cycles are required for results to converge, and much bookkeeping is required to obtain enough information for identifying performance bottlenecks.

Finally, a third class models a concurrent system as a marked graph and analyzes it as a Markov process. Kudva [14] et al. first proposed the use of this model for analyzing the performance of asynchronous circuits. Xie and Beerel [23] used symbolic techniques with this model and gave bounds on average time separation of events.

The method proposed in this paper falls into the third class. However, there are some key differences between our method and previous work. First, this paper shows that the state transitions of a system modeled by a strongly-connected marked graph exhibit an inherent periodic structure, and the system is analyzed as a *periodic* Markov chain. Previous approaches do not take into account the inherent periodic property of the system, and the system is analyzed as *aperiodic*. Theoretically, one cannot properly adopt an aperiodic Markov model to describe a periodic system, or the system would not converge. Second, by exploiting the periodicity of the system, we derive an algorithm to generate an exact, tight state space of the system, and exploit the regularity of its structure for efficient memory management and to reduce the complexity of computation. In contrast, previous approaches can generate a large number of unreachable (and unnecessary) states, resulting in excessive memory usage and runtime. Third, our approach is targeted to generating the probability distribution of input arrival time to components, as well as component utilization, as metrics for performance. The approach in [23] targets the time separation of events. Though it is also possible for our tool to generate metrics based on time separation of events, we believe input arrival time gives more directly useful information for subsequent system optimization.

There also exists a large body of work which analyses the timing behavior of asynchronous systems using *fixed* delays. Burns [3] uses weighted averages to compute expected time separation between two events where there are several possible paths of execution. Hulgaard et al. [11], Chakraborty et al. [4], and Walkup [22] give bounds on the minimum and maximum delays between time separation of events. The main application of these approaches is in timing verification, and their goal is to make sure that the *worst-case* paths over all possible execution meets system requirement. These models are unsuitable for asymptotic *average-case* analysis, which is important for system-level performance evaluation.

The key contributions of this paper are as follows. First, we provide a proof that the state transitions in decision-free asynchronous systems exhibit an inherent periodic structure.[2] To the best of our knowledge, this is the first time this property has been exploited in analyzing the performance of such systems. Second, we present a new algorithm to construct the precise reachable state space of the system. Third, we propose the use of the probability distribution of input arrival time as a metric for performance optimization. Finally, we present a practical tool to demonstrate the feasibility our approach. Initial experimental results are promising, showing several orders of magnitude improvement in both runtime and the size of the state space over previously published results.

While the focus of this paper is on asynchronous digital systems, we believe the generality of the underlying model allows the method to be applied to a large class of concurrent systems that can also be viewed as a set of independent processes that interact with each other through local synchronization. These include GALS (Globally-Asynchronous, Locally-Synchronous systems) [2, 5, 12, 16], which are systems with lower-level clocked components that communicate with each other using asynchronous handshake protocols at the higher level, as well as concurrent embedded systems.

The rest of this paper is organized as follows. Section 2 introduces two system-modeling formalisms that are used for subsequent analysis. Section 3 presents a key result of this paper: given any decision-free asynchronous system modeled as a marked graph, we prove the existence of a periodic structure in its state transition relation. Section 4 then presents the second key contribution: an algorithm to construct the entire state space of the system. Section 5 describes an efficient method to obtain useful performance analysis metrics for the given model. Section 6 then presents some experimental results and discussions. Section 7 briefly discusses some of the modeling assumptions made during the analysis and how to handle more complicated systems. Finally, Section 8 gives a brief conclusion and pointers to future work.

2. BACKGROUND: SYSTEM MODELING

This section presents two modeling formalisms: *marked graphs* and *Markov chains*. Marked graphs are used to model concurrency and data-dependent relationships between interacting components in a concurrent system. They specify the reachable states of the system and their causality. The sequence of states entered by the system as it evolves in time and their transition probabilities is captured as a Markov chain.

2.1 Marked graphs

A marked graph is a triple $N = (E, T, F)$ where E is the set of places, T the set of transitions, and $F \subseteq (E \times T) \cup (T \times E)$ the flow relation. In a marked graph, every place has at most one input and one output transition. A transition is enabled to fire whenever there is a token in each of its input places. An enabled transition fires by removing one token from each of its input places, and depositing a token in each of its output places. A *marking* is an assignment of tokens to the places in the graph. A marked graph is *safe* if in any marking reachable from an initial marking M_o, every place contains no more than one token. A marked graph is *live* if every transition is fireable, or can be made fireable through some sequence of firings from the initial marking M_o.

In the context of discrete-event systems, the marking of a marked graph corresponds to the *state* of the concurrent system, and the firing of a transition corresponds to the occurrence of an *event*.

Example: Figure 1 shows the the control circuit for a three-stage asynchronous micropipeline [20] proposed by Sutherland. Figures 2a and 2b show its corresponding marked graph models, representing two views: the former at a more detailed level with some low-level circuit components,[3] and the latter at a more abstract level. In the circuit of Figure 1, an "R" signal represents a request for a pipeline stage to process new data, and an "A" signal represents an acknowledgment to the previous pipeline stage that the data has been received. The C-elements are hardware synchronization "join" elements, which merge two threads of communication: they ensure that a pipeline stage has received a request from the previous stage as well as an acknowledgment from the next stage

[2] [11] and [24] also exploited the existence of repeating structures in marked graphs in their approaches. However, the repeating structure they used is related to the *events* in the graph structure, rather than to the *states* in the dynamic behavior of the model.

[3] This design is provided simply as an example; the reader does not need to understand details of the hardware implementation.

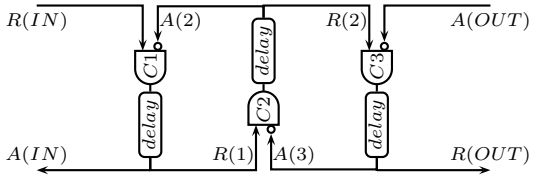

Figure 1: Control circuit for a micropipeline [20]

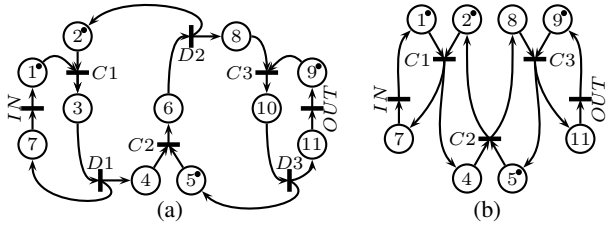

Figure 2: Marked graph model for a micropipeline [20] (a) low level (b) abstract level

before processing new data. The elements labeled "delay" represent logic processing units, whose functional details are abstracted out. Each control (C-element) and data (logic processing) unit in the circuit diagram is modeled as a transition in the marked graph. In the marked graph, a transition is represented by a rectangle, a place by a numbered circle, and a token by a dot inside a place.

2.2 Markov chains

The causal relation between state transitions in a marked graph can be captured in an intermediate specification called a *state transition graph (STG)*, which consists of a set of states S, and a state transition function $\delta : S \rightarrow S$. A Markov chain captures all the information in a state transition graph, and in addition, the transition probability between states. The information can be presented in the form of a *Markov transition matrix*, which specifies the probability $P(i, j)$ of the system making a transition from state i to state j. The transition probability between states is related to the delays of individual components of the system, which correspond to transitions in the marked graph model, and are assumed to take on random values. Random delays at each transition are modeled by independent random variables with exponential distributions. For a transition with exponentially distributed delays, the probability that it fires no later than time t after it is enabled to fire is:

$$P\{X_n \leq t\} = 1 - e^{-\mu t}, t \geq 0. \quad (1)$$

where μ is the service rate at the transition.

More than one transition can be enabled to fire in any particular state, but only one is allowed to fire (because the time between successive firings is continuous). The probability that transition T_j fires among all enabled transitions $T_1, T_2, ...T_n$ in a any particular state is given by:

$$P(T_j) = \frac{\mu_j}{\sum_{i=0}^{n} \mu_i} \quad (2)$$

where μ_i is the mean service rate at each transition.

It is also possible to assign a discrete delay distribution to a component on top of the exponential distribution, for example a component can be assigned an exponential delay of mean μ_1 50% of the time, and an exponential delay of mean μ_2 50% of the time.

The sequence of markings of the marked graph thus is described by a Markov chain subordinated to a Poisson process. The evolution of the system is described by the following transition function:

$$P(i, j, t) = \sum_{n=0}^{\infty} \frac{e^{-\mu_i t}(\mu_i t)^n}{n!} K^n(i, j), t \geq 0 \quad (3)$$

where K is the transition matrix of the underlying Markov chain, and K^n is the n-step transition probability.

3. PERIODICITY OF STRONGLY-CONNECTED MARKED GRAPHS

This section presents the first new research result: given a system which is modeled by a strongly-connected marked graph, the corresponding STG *always* exhibits a periodic behavior.

The following three theorems were proved by Commoner et al. [7] for marked graphs:

THEOREM 1. *A given marking of a graph is live and safe iff every simple cycle has exactly one token, and through every place in the graph there is a simple cycle of token count one. A marking which is live remains live after firing.*

THEOREM 2. *For every finite, directed, strongly-connected graph there exists a live, safe marking.*

THEOREM 3. *Let M be a live marking of a strongly-connected marked graph, then for any firing sequence that leads back to the initial marking M, all i transitions have been fired an equal number of times. Furthermore, there exists a firing sequence leading from M to itself, in which every transition fires exactly once.*

Intuitively, Theorem 1 indicates that the marked graph can be viewed as a set of interacting simple cycles; a marking can only be live if there is at least one token circulating on each cycle, and can only be safe if each cycle has at most one token. The next two theorems address the existence of such a live safe marking, and the steady-state regular behavior of marked graphs with live markings.

The key result that is proved in this section is that any strongly-connected marked graph corresponds to an STG with periodic structure. This result is formalized in the following proposition:

PROPOSITION 1. ***Periodic STG for strongly-connected MG*** *Given any strongly-connected marked graph, its reachable set S of all live, safe markings (states) can be divided into δ disjoint sets $B_1, B_2, .., B_\delta$ such that for any state i, j in S, and if state i belongs to set B_k, i can make a transition to j iff j belongs to the immediate next set B_{k+1}. More formally, $P(i, j) = 0$ unless $i \in B_k$ and $j \in B_{k+1}$ for $k \in 1, 2, ..., \delta - 1$, or $i \in B_\delta$ and $j \in B_1$, where $P(i, j)$ is the probability of the system making a transition from state i to state j in one transition. In other words, after a state in set B_i is visited, it always takes δ firings for a state in the same set to be visited again. Such an STG is called a **periodic** STG, and its **period** is δ.*

Proof. Let s_i be a state in a system modeled by a strongly-connected marked graph, which corresponds to a live, safe marking of the marked graph, and let B_{i+1} be the set of states reachable from s_i in one transition, and further let B_{i+2} be the set of states reachable from the states in B_{i+1} within one transition. From Theorem 3, state s_i will be visited again only after $m\delta$ transitions, and any state in B_{i+1} and B_{i+2} will be visited again only after $n\delta + 1$ and $l\delta + 2$ transitions, respectively, where m, n, l are any integers > 0. All reachable states in the systems can thus be partitioned in such a way into δ sets, where each a state in B_i can be reached from a state in B_{i-1} within one transition. □

Figure 3 illustrates the intuitive idea behind Proposition 1. We define a *synchronization point* as a transition in a marked graph with more than one input place. Figure 3a shows an example of a marked graph with a single synchronization point, which is defined as a *simple unit*. Figure 3b shows the corresponding STG of the marked graph. States in the same row of the STG form a set which can be reached through one firing only from states from the row above, and can reach through one firing only states from the row below. Intuitively, the synchronization point acts as a "gate" to constrain the movement of tokens in the marked graph, as it must wait for all its input tokens to arrive before it can fire. This gives a structural pattern to the transition of states.

Two corollaries follow immediately from Proposition 1. Corollary 1 shows that the periodicity of the state transitions in a strongly-connected marked graph is preserved through the composition of

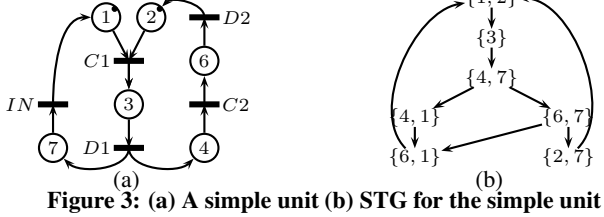

Figure 3: (a) A simple unit (b) STG for the simple unit

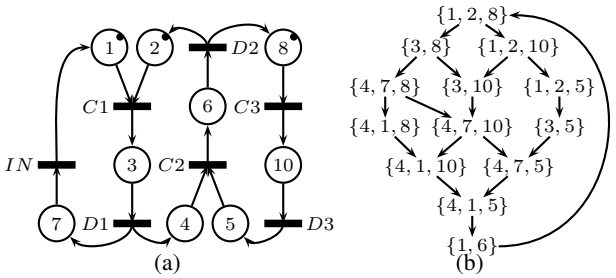

Figure 4: (a) Two coupled simple units (b) STG for the coupled units

two such graphs. This property will be used in Section 4 to derive an algorithm to construct the state space of the system.

Two marked graphs are said to be "composed together" if they are joined at one or more transitions. An example is shown in Figure 4a, where two simple units are composed at transitions $C1$ and $C2$. The corresponding STG is shown in Figure 4b.

COROLLARY 1. *Given two marked graphs with periodic STG's, the STG of their composition is also periodic.*

Proof. It follows from Theorem 2 and Proposition 1 that if the STG of a marked graph is periodic, than the underlying graph is strongly connected. Since a marked graph formed by two individual strongly-connected marked graphs composed together at one or more transitions is also strong-connected, it follows from Proposition 1 that its STG is periodic. □

The next corollary shows that the state transitions in a system modeled by a strongly-connected marked graph can be represented in a canonical form. This result will be used in Section 5 to efficiently solve for the stationary distribution of the resulting Markov transition matrix.

COROLLARY 2. *The transition matrix of the underlying Markov chain of a strongly-connected marked graph can always be expressed in the following canonical form, where δ is the period, and $B_1, B_2, ..., B_\delta$ are disjoint sets of states.*

$$P = \begin{pmatrix} 0 & B_1 & . & . & 0 \\ . & & . & & . \\ . & & & . & . \\ 0 & & & & B_{\delta-1} \\ B_\delta & 0 & . & . & 0 \end{pmatrix}$$

4. CONSTRUCTING THE STG: AN ALGORITHM

Now that it has been shown that an STG derived from any strongly-connected marked graph exhibits a periodic structure, this section presents an algorithm for constructing the periodic STG itself. The algorithm has two operators: *decomposition* and *composition*. The decomposition operator decomposes a marked graph into simple units with single synchronization points. The algorithm then constructs an STG for each of the simple units. Note that, from Proposition 1, the STG's of these simple units are periodic. The composition operator combines the periodic STG's from two intersecting simple units into a single STG. The new STG is again periodic according to Corollary 1.

The entire state space of a live, safe marked graph can thus be built by first applying the decomposition operator, and then the binary composition operator on all decomposed units recursively. Figures 5 and 6 show the pseudo-code for the decomposition and composition operators, respectively. The state space constructed using this algorithm is tight: it includes only states that are reachable from an initial live, safe marking, and no others. No extra reachability analysis steps such as that used in [23] is required. Furthermore, the constructed state space can immediately be put into the canonical form shown in Corollary 2.

```
decompose(marked_graph)
  foreach sync_pt in marked graph
    // find all simple cycles that goes through sync_pt
    foreach output_place at sync_pt
      find simple path that leads back to sync_pt
    // build state transition graph
    i = 0; // counts period
    s[0][0] = concat(output places at sync_pt)
    do until (next_transition == sync_pt)
      i++;
      j=0;
      foreach enabled_transition in current_state
        fire;
        s[i][j] = new_state
        j++;
```

Figure 5: decompose

```
compose(stg1, stg2)
  foreach state s1 in stg1
    foreach state s2 in stg2
      if s1 and s2 shares common places
        combine into new state and put in new_stg
      else if s1 and s2 do not share common places
        put state into single_state_vec
  while (single_state_vec != empty) do
    foreach state ss in single_state_vec
      foreach state ns in new_stg
        if ss and nw share common places
          combine into new state and put in new_stg
  return new_stg
```

Figure 6: compose

The intuition behind the algorithm is as follows. Since the marked graph is strongly connected, it can be broken down into a finite number of simple cycles, each of which with a token circling around it, according to Theorem 1 in Section 3. Each of these simple cycles can then be viewed as a oscillator, with a "period" equal to the number of transitions in the cycle. The composition of two simple cycles is then analogous to the coupling of two oscillators. The two coupled oscillators form a new oscillatory system, with a new period determined by the number of transitions in the new system, according to Corollary 1. The number of transitions in the new system is in turn determined by how "strong" the coupling between the two oscillators are: the stronger the coupling (meaning the more transitions they share in common), the shorter is the period of the resulting coupled system.

Another interesting observation that can be made on the algorithm is that once the STG's of the decomposed simple units are constructed, the STG of the entire design space is composed by simply "stitching" the lower level STG's together recursively. In other words, the state space of the entire system is constructed based solely on the structure of marked graph. There is no notion of the dynamic behavior of the system during the construction, i.e., the actual marking plays no role in the analysis. In contrast, most existing algorithms for state space exploration requires keeping track of a set of current states and computing a set of legal next states through some search strategy [8]. The complexity of our algorithm is considerably lower.

5. OBTAINING PERFORMANCE ANALYSIS RESULTS

The result of the previous algorithm is a periodic STG, which can be transformed into a periodic Markov transition matrix according to the relationship outlined in Section 2.2. From the transition ma-

trix of a Markov chain, one can obtain the asymptotic behavior of a system by solving the matrix for its *stationary distribution*. Techniques for finding the stationary distribution of Markov chains can be found in standard text books on stochastic processes such as [6] and [9]. We summarize some important results for solving periodic Markov chains below.

Let P be the transition matrix of an irreducible Markov chain with recurrent periodic states of period δ. Then the transition matrix $\bar{P} = P^{\delta}$ can expressed as:

$$\bar{P} = \begin{pmatrix} P_1 & & & 0 \\ & P_2 & & \\ & & \cdot & \\ & & & \cdot \\ 0 & & & P_\delta \end{pmatrix}$$

The sub-matrices $P_1, P_2, ...P_\delta$ form δ closed sets, each of which is irreducible, recurrent and aperiodic. A Markov chain is *irreducible* if and only if all states can be reached from each other. A Markov chain is *recurrent* if all states can be visited infinitely often.

The stationary distribution $\pi_1, \pi_2, ..., \pi_\delta$ for each of the sub-matrices can be solved for and is given by the following set of equations:

$$\pi_{\mathbf{k}} P_k = \pi_{\mathbf{k}} \quad (4)$$

$$\pi_{\mathbf{k}} \mathbf{1} = \mathbf{1} \qquad \text{for } k = 1, 2, ..., \delta \quad (5)$$

$P(i,j)$ itself does not have a stationary distribution. However, the limit of $P^{n\delta+m}(i,j)$ exists as $n \to \infty$ but are dependent on the initial state i.

$$\lim_{n\to\infty} P^{n\delta+m}(i,j) = \begin{cases} \pi(j) \text{ if } i \in B_\alpha,\ j \in B_\beta, \\ \qquad \beta = \alpha + m(mod\ \delta); \\ 0 \text{ otherwise} \end{cases}$$

Once the stationary distribution for each sub-matrix is found, useful performance metrics can be obtained as guides for optimization. For example, the stationary state distribution gives direct information on *input arrival time*, i.e., the probability of each input to a component arriving last (or first). System bottlenecks can be identified as paths in the system that lead to a late input arrivals with a high probability. Consider the micropipeline example in Figure 1 and its marked graph model in Figure 2a. Suppose that the stationary probabilities of the system being in a state where there is a token in input place 4 but not in input place 5, and that of a token in input place 5 but not in input place 4, are 0.1 and 0.5, respectively. From this one can deduce that the component C2 is five times more likely to wait for a data input from input place 4 than from input place 5. With this information, one can optimize the performance of the micropipeline by speeding up the critical path, for example, by using a faster function unit in stage 1 (i.e., at D1), or through transistor sizing to speed up the input, R(1), to C2.

Component utilization can be obtained by computing the probability of the component being in a state waiting for one or more of its inputs to arrive, divided by the probability of it being in a state where all input tokens are present. *System latency*, i.e., the average time separation between a given input event and output event of the system, can be found by computing the weighted average of all paths of executions of the corresponding states in the STG. Similarly, *system throughput* can be found by computing the average time separation between two events of the same output in successive iterations of the system.

Figure 7 shows a summary of the performance analysis flow.

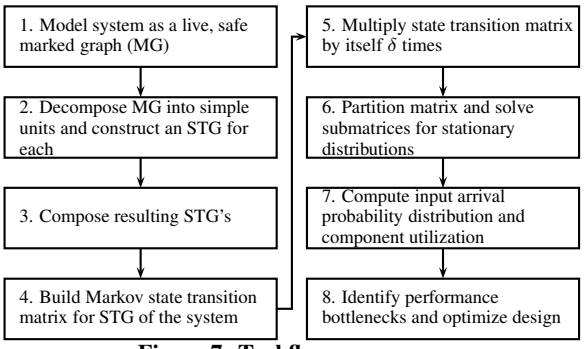

Figure 7: Tool flow summary

test case	design instance	# reachable states	CPU Time (seconds)
micropipeline [20]	3-stage	29	0.003
	4-stage	70	0.007
	5-stage	169	0.031
	6-stage	408	0.164
	7-stage	985	0.986
	8-stage	2377	7.820
	9-stage	5740	29.835
	10-stage	13859	361.621
	11-stage	33460	4686.126
Huffman decoder [1]		160	0.036
DiffEq [25]		175	0.039
DCT [21]		169	0.031

Table 1: Experimental Results

6. TEST CASES AND DISCUSSION

A tool has been implemented to demonstrate the performance analysis method proposed in this paper. Table 1 shows a summary of results for a number test cases ran on an Intel Xeon CPU at 3.06GHz with 1GB of memory.

The first set of test cases are different variants of Sutherland's micropipeline design [20] shown in Figure 1. The second test case is an asynchronous Huffman decoder design proposed in [1]. The third test case is a low-control-overhead asynchronous differential equation solver proposed in [25], and the fourth is an asynchronous DCT matrix-vector multiplier proposed in [21]. In each testcase, the architectural flow diagram is converted to a marked graph to capture data dependency between functional unit and concurrency.

In [23], results were reported for a 6-stage pipeline design similar to our test case 1, and reported reachable states of up to 28,000 and runtime of just less than one hour on a SPARC 20. As a followup, in [24], it was indicated that the approach in [23] cannot handle pipeline designs of more than 8 stages. Our result shows significant improvements over their reported results. The performance results for an 8-stage pipeline can be obtained in 7.8 seconds. While different computing environment precludes any meaningful comparison in runtime, we would like to point out that, first, our state space is much smaller, as our algorithm prunes all unreachable and transient states from analysis. Second, the matrices we solve are also smaller, as their average size is the total number of states divided by the period of the system. Third, a stationary distribution for each of the sub-matrices in our analysis is guaranteed to exist. In fact, without our pruning of unreachable and transient states, and without taking into account the periodicity of the system, a stationary distribution for the system theoretically does not exist.

A few other notes can be made on the results. First, the size of the state space is exponential in the number of parallel operations that can run in the system at the same time. When applied to real designs, this result is not as pessimistic as it seems, as the number of process running in parallel is often limited. A full-buffer pipeline where each stage performs a different operation represent the worst case scenario, while many highly-concurrent systems have more moderate amounts of concurrency.

Secondly, while prior work [23] reported the solving for the stationary distribution of the transition matrix of the Markov chain as the bottleneck of the analysis, the same operation does not present a problem in our analysis. In fact, profiling shows the runtime is dominated by the matrix multiplication in Step 5 of the tool flow. Matrix multiplication is an $O(n^3)$ operation in general. Due to the regular structure of the matrix in our application, the multiplication can be simplified, and the complexity is $O(m^3)$, where m is the size of the largest sub-matrix of the system. The memory require-

ment for storing the matrix is greatly reduced as well, as only the submatrices need to be stored.

7. MODELING ISSUES

This section addresses two limitations of the model used in this paper: the use of an exponential delay distribution, and the restriction to decision-free systems.

In order to make a system amenable to analytical methods, the delay distributions for components are currently restricted to be exponential. In reality, delays in computation seldom resemble an exponential distribution. This restriction, however, should not have a significant impact on results in practice. As shown in [17], simulation results with delay distributions from real life data are often very close to those of an exponential distribution. Only the mean and the variance of delay have a significant impact on the result. This, indeed, is consistent with experiences from the performance analysis community. For component delays with relatively small variance compared to its mean (precisely, if the value of the variance is smaller the value of the mean), an exponential distribution serves as a sufficient worst-case bound for performance. Component delays with large variances can be approximated as discrete distributions, as shown in Section 2.2. It should be noted that while the use of exact delays is important in some applications such as timing verification, it is not as important in performance analysis.

Another apparent drawback of our approach is the restriction to decision-free systems, as systems with complicated choice decisions and conflicts cannot be properly modeled as marked graphs. With proper modeling, however, systems with simple choices can be analyzed under our framework. For example, a choice between a "slow mode" and a "fast mode" of operation of a certain component can be modeled using a discrete delay distribution. In some cases, more complicated choices can be handled hierarchically. Extensions to handle arbitrary systems with choice in a more integrated way will be included in future work.

8. CONCLUSIONS AND FUTURE WORK

We have presented an efficient solution for in analyzing the asymptotic performance of asynchronous systems.

We model an asynchronous system as a marked graph, and capture its underlying state transition as a Markov chain. We showed that the state transition graph of a system modeled by a marked graph exhibits a periodic structure, and proposed an algorithm for constructing a tight state space of the system based on this property, which is then transformed to a Markov chain. Local asymptotic performance metrics, such as *distribution of input arrival time* and *component utilization*, are obtained by solving the transition matrix of the Markov chain for its stationary state distribution. Using this information, global metrics for system bottleneck and cycle slack can then be derived, which can in turn be used to in the future as a guide for system-level optimization, such as through cycle balancing. We demonstrated our method via a tool. Experimental results show significant improvement over previously published results in terms of both the size of state space and run time.

Our proposed approach effectively views an asynchronous system as cyclical, compositional and recursive in structure, and periodic in its dynamics. This view has facilitated us to derive an efficient method to analyze system performance, and to define meaningful performance metrics for optimization. In contrast, existing work on the performance analysis of asynchronous systems often views the cyclic structure of these systems as an undesirable property and seek to analyze them as acyclic by unfolding their corresponding marked graph model. We have offered a new, and in our view, potentially more suitable, way of looking at asynchronous systems, which we believe would lead to our ultimate goal of building optimal systems. In more detail, our compositional method for constructing the state space of the system under investigation based on their periodic property is analogous to the coupling of a system of oscillators. Oscillator models have been used for modeling various systems in the natural sciences and in engineering, for example, in robotics and in distributing clock signals in clocked systems [18]. We believe that investigating this connection further can lead to interesting methods for engineering the design and synthesis of efficient asynchronous systems.

We see many avenues for further investigation. Research goals in the immediate future include extensions to analyze asynchronous systems with choice, the development of performance optimization algorithms for asynchronous systems driven by our analysis technique, and the application of our method to a broader class of concurrent systems, such as GALS and embedded systems.

Acknowledgments

The authors thank Prof. Stephen Edwards of Columbia University for background on synchronous modeling languages.

9. REFERENCES

[1] M. Benes, S. Nowick, and A. Wolfe. A fast asynchronous huffman decoder for compressed-code embedded processors. In *Proceedings of ASYNC*, 1998.

[2] D. S. Bormann and P. Y. K. Cheung. Asynchronous wrapper for heterogeneous systems. In *Proceedings of ICCD*, 1997.

[3] S. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, Caltech, 1991.

[4] S. Chakraborty and R. Angrish. Probabilistic timing analysis of asynchronous systems with moments of delays. In *Proceedings of ASYNC*, 2002.

[5] D. Chapiro. *Globally-asynchronous, locally-synchronous systems*. PhD thesis, Stanford University, 1984.

[6] E. Cinlar. *Introduction to Stochastic Processes*. Prentice-Hall, Inc., 1973.

[7] F. Commoner, A. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[8] G. de Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[9] W. Feller. *An Introduction to Probability Theory and Its Application, Vol. 1*. John Wiley and Sons, Inc., 1968.

[10] M. Greenstreet and K. Steiglitz. Bubbles can make self-timed pipelines fast. *Journal of VLSI Signal Processing*, 2:139–148, 1990.

[11] H. Hulgaard, S. M. Burns, T. Amon, and G. Borrielo. An algorithm for exact bounds on the time separation of events in concurrent systems. In *IEEE Transactions on Computers*, 1995.

[12] A. Iyer and D. Marculescu. Power-performance evaluation of globally asynchronous, locally synchronous processors. In *Proceedings of the 29th International Symposium on Computer Architectures*, 2002.

[13] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, Aug. 1974. North-Holland.

[14] P. Kudva, G. Gopalakrishnan, E. Brunvand, and V. Akella. Performance analysis and optimization of aysnchronous circuits. In *Proceedings of ICCD*, 1994.

[15] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987.

[16] S. Moore, G. Taylor, R. Mullins, and P. Robinson. Point to point GALS interconnect. In *Proceedings of ASYNC*, 2002.

[17] P. B. Pang and M. Greenstreet. Self-timed meshes are faster than synchronous. In *Proceedings of ASYNC*, 1997.

[18] G. Pratt and J. Nguyen. Distributed synchronous clocking. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):314–328, 1995.

[19] S. Rotem, K. Stevens, B. Agapiev, C. Dike, M. Roncken, R. Ginosor, R. Kol, P. A. Beerel, K. Y. Yun, and C. J. Myers. Rappid: An asynchronous instruction length decoding and steering unit. In *Proceedings of ASYNC*, 1999.

[20] I. Sutherland. Micropipelines. In *Communications of the ACM*, 1989.

[21] S. Tugsinavisut, Y. Hong, D. Kim, K. Kim, and P. Beerel. Efficient asynchronous bundled-data pipelines for dct matrix-vector multiplication. *IEEE Transactions on VLSI*, 13(4):448–461, 2005.

[22] E. Walkup. *Optimization of Linear Max-Plus Systems with Application to Timing Analysis*. PhD thesis, University of Washington, 1995.

[23] A. Xie and P. A. Beerel. Symbolic techniques for performance analysis of asynchronous systems based on average time separation of events. In *Proceedings of ASYNC*, 1997.

[24] A. Xie, S. Kim, and P. A. Beerel. Bounding average time separations of events in stochastic timed petri nets with choice. In *Proceedings of ASYNC*, 1999.

[25] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. E. Dooply, and J. Arceo. A low-control-overhead asynchronous differential equation solver. In *Proceedings of ASYNC*, 1997.