Performance Analysis and Optimization of Asynchronous Circuits

PRABHAKAR KUDVA (*pkudva@cs.utah.edu*)
GANESH GOPALAKRISHNAN* (*ganesh@cs.utah.edu*)
ERIK BRUNVAND* (*elb@cs.utah.edu*)

Dept. of Computer Science
University of Utah
Salt Lake City, Utah 84112

Keywords: performance analysis, self-timed circuits, asynchronous systems, timed petri-nets

Abstract. Asynchronous/Self-timed circuits are beginning to attract renewed attention as promising means of dealing with the complexity of modern VLSI designs. However, there are very few analysis techniques or tools available for estimating the performance of asynchronous circuits. In this paper we adapt the theory of Generalized Timed Petri-nets (GTPN) for analyzing and comparing a wide variety of asynchronous circuits, ranging from purely control-oriented circuits such as cross-bar arbiters to large asynchronous systems with data dependent control such as asynchronous processors. Experiments with the GTPN analyzer are found to track the observed performance of actual asynchronous circuits, thereby offering empirical evidence towards the soundness of the modeling approach. Our main contribution is in demonstrating how a *quantitative design methodology* for asynchronous circuits can be developed based on Timed Petri-nets.

1 Introduction

Asynchronous/Self-timed designs are beginning to attract renewed attention as promising means of dealing with the complexity of modern VLSI designs. The advantages of self-timed systems include: (i) freedom from problems associated reliable global-clock generation and distribution and the associated power wastage in the clock drivers; (ii) modularity due to the absence of global control schedules, which makes asynchronous circuits easier to modify incrementally; and (iii) their capacity to exhibit better average case performance, as they do not have to wait for the next clock "tick" to arrive before a following operation can be triggered [22, 34, 12]. Asynchronous circuits are often claimed to exhibit better average case performance (here, and elsewhere, performance means *overall speed* or *throughput*). However, very few attempts have been made to study the problem of estimating the performance of a wide variety of asynchronous circuits as well as asynchronous systems, to understand why such a claim could be true, and the circumstances under which asynchronous circuits could perform better. In this

*Supported in part by NSF Award MIP-9215878

paper, we propose performance analysis techniques for asynchronous *self-timed* [22] circuits and systems based on timed Petri nets. One of the important features of our work is that our analysis technique can handle circuits whose execution time depends on the data being manipulated by the circuit, as well as deal with arbitration and resource usages.

It is in a way ironical that the mechanism that is claimed to be responsible for the improved average-case performance of asynchronous circuits—the absence of a global clock—is also responsible for making the problem of analyzing the performance of asynchronous circuits harder. Since synchronous clocks couple the computation- and physical-time, performance evaluation in the area of synchronous system design is almost always carried out in terms of the clock cycle time [7, 15]. Since there is no such global measure of time in asynchronous circuits, the problem is much more difficult. In addition, asynchronous circuits are often used to realize systems such as high-performance communication front-ends [9], high-performance cache memory subsystems [27], complex arbiter circuits [14], high performance arithmetic units [12], and decoupled processors [13, 5] whose execution time is data dependent. The execution characteristics of these systems are very decoupled as well as probabilistic in nature. Arbitration for shared resources is an important concern in most of these systems. Therefore, in order to get a handle on the problem of analyzing the performance of asynchronous circuits, we must select a suitable formalism that can deal with the above mentioned situations. *Timed Petri nets* can model systems with these characteristics, and therefore are central to our approach.

1.1 Related Work and Overview of Our Approach

Many of the efforts in the area of asynchronous circuits are geared towards the study of synthesis techniques, correctness issues, etc., and are not so much geared towards understanding performance aspects of asynchronous systems. Asynchronous circuit performance analysis has been conducted at an analytical level by Sparsoe [32] and Williams [36]. Their work is primarily concerned with the study of asynchronous pipelines. They do not consider data dependent behaviors, arbitration, or resource usages.

Petri Nets have been widely used for studying asynchronous computations [25] as well as for modeling as well as synthesizing synchronous [29] and asynchronous circuits [8, 3]. They have also been widely used in performance analysis studies at the system level. Basically one can observe two categories of works in this area. In the first category, decision-free Petri net structures are used to model computations [30] while in the second category, Petri nets with decisions (or "choices") are allowed [37, 18]. Recent work on asynchronous circuit performance analysis falling under the first category include the works of Burns and and Williams [6, 36]. These authors model the computation as a constraint-graph and solve

performance equations using linear programming techniques. Work by Hulgaard et al. [16] adapt the work by Burns and provide an algorithm to find exact bounds on the time separation of events in a process graph without conditional behavior which serves as a basis for evaluation during time constrained synthesis or design. Nielson and Kishinevsky [26] address the problem of determining the cycle time and critical paths using the timing simulation of an underlying Signal graph. These techniques are limited in their applicability to branching behaviors, especially those dependent on data statistics, which are an important part of the claimed average-case performance advantages of asynchronous circuits.

To the best of our knowledge, there have been no attempts so far at studying the performance of asynchronous circuits that exploit data dependent branchings—using timed Petri nets or any other formalism. Our choice of timed Petri nets as the internal representation has the added advantage that we can later interface the performance analysis tool with the high level synthesis tool developed by our group [3]. The main contribution of this paper is to demonstrate that using an existing Generalized Timed Petri Net (GTPN) model [18] (detailed in Section 3), it is possible to model a wide class of asynchronous circuits and systems in a uniform manner, model arbitration as well as resource usages, model circuits hierarchically, and thereby make insightful observations about factors that determine the performance of asynchronous circuits.

1.2 Outline of the Remainder of the Paper

In Section 2, we present how we model self-timed circuits using Petri nets. In Section 3, we present an overview of the theory of Generalized Timed Petri nets and discuss the performance measures used in this paper. In Section 4, we provide three examples to illustrate the power of our technique: a crossbar arbiter circuit, an asynchronous reordering buffer, and the execution unit of an asynchronous processor. Section 5 provides concluding remarks.

2 Self-timed Circuits and their Representation

A self-timed element (or module) is a piece of hardware which accepts a signal on its input corresponding to initiation of a specific computation and produces a signal on its output to denote the completion of the computation. A self-timed circuit is a *legal* interconnection of self-timed elements. A wide class of modern self-timed circuits *roughly* fall in this category [11, 34, 4, 2, 22]. We assume two-phase transition signaling (up-going and down-going transition of a signal have the same meaning) and assume one wire per bit of data with the data-bundling assumption, *i.e.*, the data is assumed to be valid at all the receivers before the corresponding control signal arrives [34]. Furthermore, we assume that

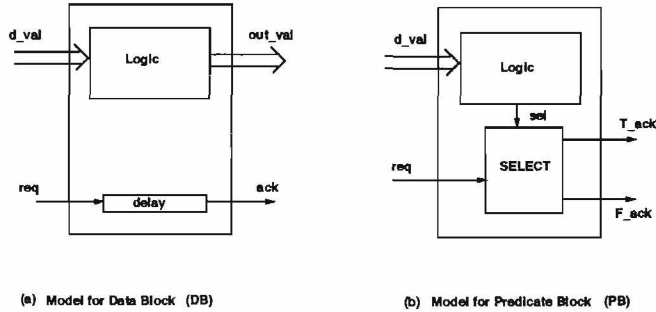

Figure 1: Models for DB and PB

the asynchronous circuits considered by our analysis procedure are built using macromodules [34, 4, 2] although, in principle, our analysis procedure should apply to circuits built using alternate design styles also.

Macromodule circuits can be classified as control blocks (CB), datapath blocks (DB), and data-dependent control modules, or predicate blocks, PB (the latter two are illustrated in Figure 1). Examples of primitive CBs include the XOR gate, the Muller C-ELEMENT, and the CALL module. In Figure 1, the delay line that is associated with the DB logic models the datapath delay. PB implements data-dependent control flow, *e.g.*, conditional branches. It employs the SELECT module which steers the transition on its *req* input to either the *T-ack* or the *F-ack* output depending on whether the logic signal *sel* is a 1 or a 0 respectively which, in turn, depends on whether d_val meets the condition being tested or not.

2.1 Petri Net Representation of Self-timed Circuits

As observed by several researchers in the past [23, 10] a self-timed circuit can be represented by a Petri net [1]. We describe each circuit element by a Petri net and express the complete circuit as a structural *composition* of the Petri nets corresponding to the individual components. Each transition (or bar) in a Petri net is labeled with a wire name. The firing of a transition represents a signal transition on the wire which labels it. Thus, the sequence of wire names in a firing sequence represents a possible history of wire transitions. It is easy to show that our Petri nets are one-safe, i.e., the maximum number of tokens in any given place is 1.

The Petri nets corresponding to circuits have a special set of places called the *initial places* (IP) and a special set of places called *final places* (FP). IP and FP are disjoint and they correspond to the inputs and outputs of the circuit. The operation of a self-timed circuit is modeled by placing tokens in

[1] we assume the reader is familiar with the semantics of a Petri net, due to paucity of space we do not repeat it here

IP (corresponds to initiating a computation) and *waiting* for tokens to appear in FP (corresponds to acknowledge). The environment (i.e. whoever is using the circuit) is responsible for placing the tokens in IP and removing the tokens from FP. We assume that the environment of a circuit uses the circuit *properly*, i.e., without causing glitching and hazards.

2.2 Composition

The netlist of a self-timed system can be described by a composition of the Petri nets corresponding to the individual circuit elements. For each module, the initial places indicate inputs to the module. Each initial place is connected to a bar which is labeled with a corresponding wire name. Similarly, each final place represents an output and is connected to a bar which is labeled by the corresponding wire name. We define the structural interconnection of two modules, where the output of one module is connected to the input of the next by making their corresponding final and initial places the same. The above composition method is illustrated on an example shown in Figure 2. The request, *Req*, and the acknowledgement, *ack*, are inputs to the C-ELEMENT. The Petri net for the C-element is one possible representation (join) for the C-element and more detailed models can be used. The output of the C element is the request to the PB block which checks the greater of the two data values A and B coming to it. If $A > B$ then, A is added to B (data block, DB); else, a new value is loaded into the register, *reg* (data block). The Petri net for this example is shown in Figure 2(a) as a composition of the Petri Nets of these self-timed modules.

2.3 Hierarchy of Petri nets

One of the problems for very large circuits and systems in using Petri nets is the explosion in the size of the Petri net resulting both in lack of clarity in the modeling of the circuit as well as an explosion in the state of the reachability graph of the net. We therefore model the Petri Net hierarchically. We introduce a hierarchical transition in a Petri Net which can be expanded into a sub Petri net. The input places of the transition are the same as the input places of the sub Petri net and the output places of the transition are the same as the output places of the sub Petri net. This way, the information from the analysis of the lower level Petri net can be used to annotate the hierarchical transition of the Petri net. This will be discussed in detail in Section 3. We will restrict the sub-Petri nets to have only a single recurrent class in its reachability graph. This is not a restriction for most hardware systems.

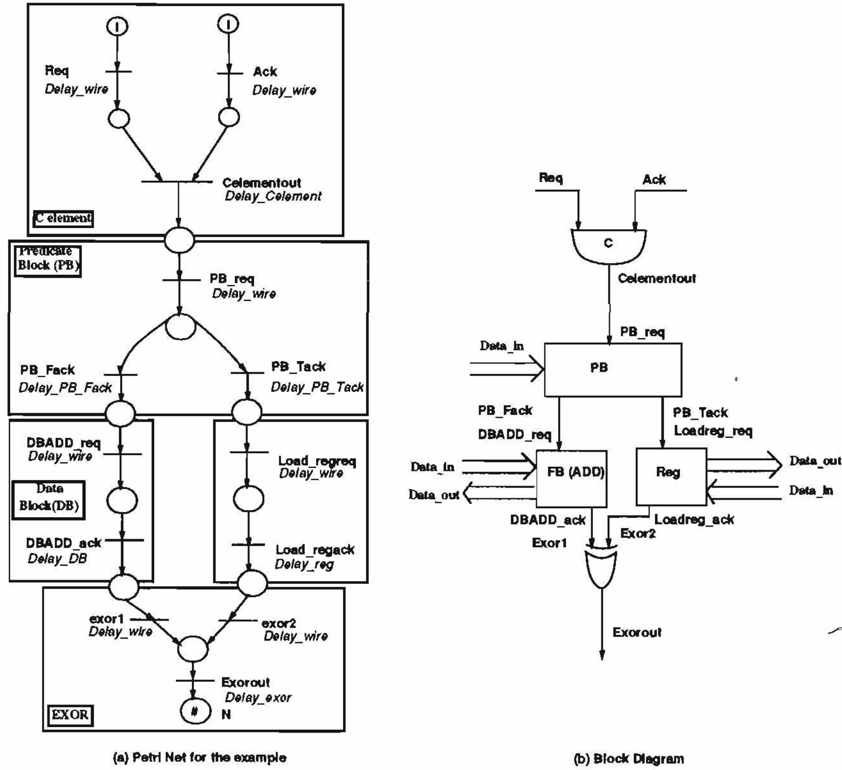

Figure 2: Example for Illustration of the Modeling and Estimation

3 Theory of Generalized Timed Petri-nets

In this section, we first present the GTPNA formalism and show how the asynchronous circuit representation is analyzed using this model. We describe existing means for analysis of the Petri Nets, and how they are adapted and extended for our purposes.

3.1 Timed Petri Nets

A GTPN [18] is a Petri net which has been augmented to include a set of firing durations (D), a set of firing frequencies (F), and a possibly empty set of resources (R) associated with each transition. Formally, $GTPN = (P, T, A, M_0, D, F, R)$, where P is the set of places, T the set of Transitions, and A is a set of directed arcs which can connect only transitions to places and places to transitions. M_0 indicates the initial marking marking of places in the Petri Net. A transition has a deterministic firing duration D associated with it. Each transition is associated with a *start firing* event and an *end firing* event in between which the firing of the transition is in progess. The removal of tokens from a transition's input places occurs at *start firing* while the placement of tokens on a transitions's output places occurs at *end*

firing. While the firing of a transition is in progress, the time to *end firing*, called the *remaining firing time* (RFT), decreases from the firing duration to zero. Thus, the state of a GTPN is only partially defined by the distribution of tokens; its state must also include the RFT of each transition that is currently firing. Each transition of a GTPN is generated by a set of start firing or a set of end firing events that occur simultaneously. The maximal set of transitions that can start firing simultaneously defines the next state of a GTPN. The *frequencies*, F, associated with each transition are used in assigning probabilities to the next state of the Petri net. (The notion of frequencies will be explained in detail in Section 3.2.) Note that the first four components of the tuple of a GTPN are identical to similar constructs of an untimed Petri Net.

3.2 Modeling of delay and firing frequencies

In our approach, the delays in the circuits are modeled by the firing durations (D) of the transitions in the Petri nets. These delays can be obtained from a library, through experiments, or through known timing estimation techniques. Additional conventions in delay modeling are now explained (and illustrated in Figure 2):

- A transition in the Petri net indicating a signal transition at the output of a gate is annotated by the delay of the gate. In this case, we do not make any distinction between rising and falling transitions, and give an average figure taking into account both rise times and fall times. This is shown by *Delay_Celement* and *Delay_exor* in Figure 2.
- A transition in the Petri net indicating a signal transition at the input of a gate is annotated by the delay of the wire from the output of the previous gate (of which it is an output) to the input of this gate. In the case of a primary input signal, we can annotate the transition with either zero or some deterministic real number indicating the delay from the primary input to the input of the gate. This is shown as *Delay_wire* next to the transitions in Figure 2.
- A transition indicating the acknowledge signal from a DB is annotated by the average case delay of the DB. In Figure 2, this is represented by *Delay_DB*
- Transitions indicating acknowledge signals from a PB are annotated by the sum of the average case delay of logic block and the delay of the Select element in generating the respective acknowledge transition. This is shown as *Delay_PB* in the example.
- In the case of a hierarchical transition, we annotate the transition with the the long run time (this will be discussed in Section 3.3) of the sub Petri net. We also note that due to the hierarchical

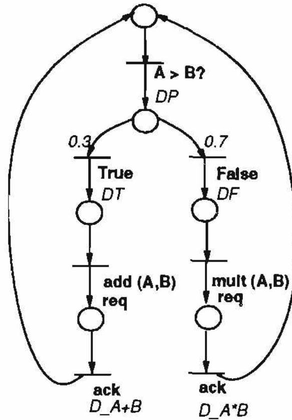

Figure 3: Simple Example to Illustrate Firing Frequencies and Long Run Time

nature of the Petri net, a transition at a higher level in the hierarchy would have more abstract system level timing information rather than detailed circuit timing delays.

The firing frequencies (F) can be assigned to any transition. In case of a place with two or more arcs leading out of it (indicating a choice), branching frequencies may be assigned based on data dependent branching information. We note that the equivalent circuit paradigm of such an occurrence in the Petri net is a PB, which is a Select element whose control signal behavior is data dependent. The examples in Figure 3 illustrate the way we assign branching frequencies. In Figure 3, we assign the probability of $A > B$ being true or false to the transitions to be 0.3 and 0.7. All other transitions are assumed to have a firing frequency of 1.

3.3 Background on the GTPNA analysis techniques

We present below a quick overview of the GTPNA analysis techniques from Holliday [18], for the sake of completeness. The following sketch of some of the relevant terminology and theorems of Markov Chain theory is presented here. A detailed discussion of the Discrete-Time Markov Chains and their relationship to stochastic Petri nets are described in [1, 24]. The strongly connected components of the state space (when viewed as a directed graph) are the *classes* of the Markov Chain. The condensed graph (one vertex for each class) is a directed acyclic graph with one root. In the case of a finite state space, the interior vertices of the condensed graph are called *transient classes*. The leaf vertices of the condensed graph are called *recurrent classes* $\mathcal{R}$. In a typical evolution of the system being modeled, the system starts in a state in the root of this condensed graph. It then filters through the transient classes until it is absorbed by one of the recurrent classes. Once absorbed it stays in that recurrent class

permanently.

In a GTPN, we are interested in the long run (or "steady state") behavior. There are two characteristics of the movement through the transient classes that are of interest: the *absorption probabilities*, or the probabilities of being absorbed by each leaf (recurrent class); and the *mean time of absorption* into the recurrent class. For techniques to find these characteristics, readers are referred to [17]. Once absorbed in a particular recurrent class, the the *long run* [20] probability distribution over the states in the recurrent class needs to be computed. This stationary probability distribution $\Pi_{\mathcal{R}}$ is easy to find since it is the unique solution to the set of equations $\Pi_{\mathcal{R}} = \Pi_{\mathcal{R}} P_{\mathcal{R}}$ and $\sum_{j \in \mathcal{R}} \Pi_j = 1$, where matrix $P_{\mathcal{R}} = \{P_{ij} | i, j \in \mathcal{R}\}$. Here, P_{ij} refers to the probability of visiting state j from state i where both states are within the recurrent class $\mathcal{R}$. The fact that $\sum_{j \in \mathcal{R}} \Pi_j = 1$ corresponds to the fact that the set of states $\mathcal{R}$ is a *recurrent* class, *i.e.*, once execution enters $\mathcal{R}$, it does not leave $\mathcal{R}$. Both P_{ij} and P_i are obtained from the underlying state graph of the timed Petri net [17]. Equation $\Pi_{\mathcal{R}} = \Pi_{\mathcal{R}} P_{\mathcal{R}}$ corresponds to the fact that in the long run, instead of having a probability of going from state i to state j in the recurrent class $\mathcal{R}$ (as defined by the transition matrix), we now have a vector of probability values $\Pi_{\mathcal{R}}$ of visiting each state i in the recurrent class $\mathcal{R}$.

From $\Pi_{\mathcal{R}}$, we can find the *long-run time* as well the *relative time spent* in each state of the recurrent class as follows. Let S be the set of states in recurrent class $\mathcal{R}$ and let s_1 be a state in S. Also, define $RelTime(s_1)$ to be the time spent in state s_1 relative to the long-run time for $\mathcal{R}$. The long-run time for recurrent class $\mathcal{R}$ for any state s_1 is defined as

$$LongRunTime(s_1) = \frac{\sum_{k \in S} TimeInState(k)\ \Pi_k}{\Pi_{s_1}} \tag{1}$$

where $TimeInState(k)$ is the time spent in state k which is determined from the user given timing annotations in the Petri net and Π_i is the probability of visiting state i in the long run. $RelTime(s_1)$ is given by

$$RelTime(s_1) = \frac{TimeInState(s_1)\ \Pi_{S_1}}{\sum_{k \in S} TimeInState(k)\ \Pi_k}. \tag{2}$$

We also can obtain for each recurrent class, the utilization of the resources while in that class, we find E[Resusages], the expectation of Resource Usages for a recurrent class is given by

$$E[ResUsages] = \sum_{k \in S} ResUsages(k)\ RelTime(k) \tag{3}$$

where $ResUsages(k)$ is the amount of resources used while in state k of the recurrent class.

3.4 Performance Estimation of Asynchronous circuits

In estimating the performance of asynchronous circuits, we employ the long-run time and the resource usage as performance measures. Our usage of long-run time in this manner is related to Zuberek's [37] use of this notion on examples with a single recurrence class.

It is important to note that this *long run time* is useful in the comparison of different designs and serves as a numerical measure with which to compare the designs. For example, Zuberek has suggested that by using this approach two architectures can be compared against each other, although he does this only for systems that can be modeled as Petri Nets with a single recurrence class. The ratio of the performance of an optimized and unoptimized design gives us a fair idea of the amount of improvement we can see in the optimization process. For example, if we want to add another instruction to the processor, we would like to know the speedup versus the cost of adding such an instruction. Another example is where we decide on multiplexing two sets of 16 bit data through the input pins of a chip, in a pin limited design. We calculate the speedup if the data was not multiplexed, i.e; 32 dedicated pins were assigned for inputting the data. The cost versus the speedup decision can then be left to the designer.

In the case that the recurrent class belongs to a Petri net which is a subnet (represented as a hierarchical transition in a higher level Petri net), the long run time becomes the firing duration (D) of the hierarchical transition. This makes it possible to apply the results of Section 3.3 for the hierarchical Petri nets as well.

4 Examples

Two examples are considered in this section. In Section 4.1, we present the analysis of a symmetric crossbar arbiter [35]. Arbiters are control intensive circuits whose performance is critical to the overall performance of the asynchronous system using them. The crossbar circuit considered here is the asynchronous version proposed in [14]. This example demonstrates the ability of GTPN to allow conditional (state dependent) probabilities. In Section 4.2, the analysis of the Non-Synchronous RISC (NSR) [5] self-timed processor architecture is presented. In the same section, we compare the performance estimated by the GTPN tool against actually measured performance figures from the NSR processor's Actel FPGA circuit, and point out that these measurements track each other, thereby justifying our approach. Additional examples are briefly discussed in Section 4.3.

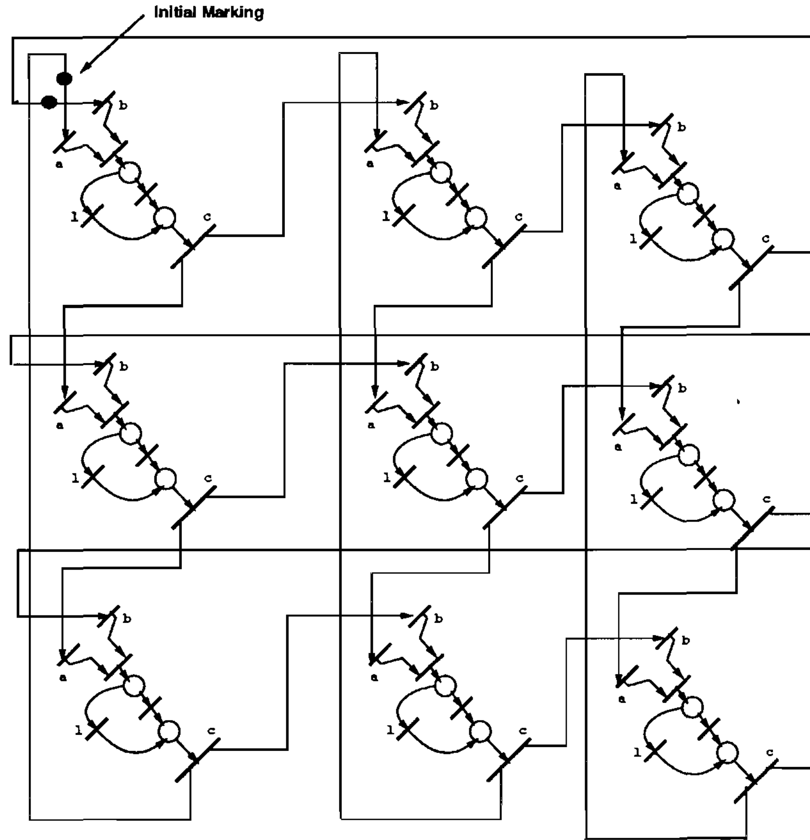

Figure 4: Petri Net Model of the Wavefront Arbiter

4.1 Symmetric Crossbar Arbiters

Consider an N×N crossbar switch that receives (and queues up) connection requests issued at random from the outside world. As an example, for a 4×4 arbiter with rows and columns numbered 1 through 4, the queue state can read (1,2), (1,3), (2,1), (4,2) at a particular time. The problem faced by a crossbar arbiter is to select, from among the queued up requests, the maximum number of requests that will not conflict on a row or a column. For our example, the maximum number of switch connections are achieved by closing (2,1), (1,3), and (4,2).

In [14], a crossbar arbiter, called the *wavefront arbiter* has been proposed. In the wavefront arbiter, a diagonal wavefront travels from the top-left corner to the bottom-right corner, sequentially examining pending requests. Whenever a request is encountered at (x, y), the portion of the wavefront at (x, y) is pinned down till the request at (x, y) has been processed. The portions of the wavefront at coordinates (x', y') where x' and y' are less than x and y are, however allowed to move forwards. In the case of our example, the wavefront arbiter can connect only locations (2,1) and (1,3) simultaneously; switch (4,2)

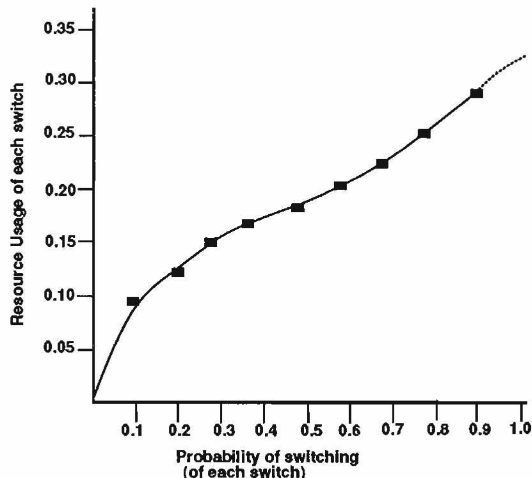

Figure 5: Usage of switches in the Wavefront Arbiter

will not be closed by it till switch (2,1) has been serviced.

The Petri-net for the wavefront arbiter is given in Figure 4. Each location of this arbiter is realized using a component called the LOCK-C [14], which is a Muller C-ELEMENT that can be asynchronously locked. According to the figure, when a and b arrive, the LOCK-C element gets enabled. While in the enabled state, the l input ("lock") can arrive, thus postponing the firing of c. When locked, the wavefront is effectively pinned down by the LOCK-C. The crossbar switch at this location can now be closed for the desired duration of time. Following that l can be deasserted when the LOCK-C is unlocked and the wavefront is allowed to proceed once again.

In the Figure 5, we show the usage of the switches (throughput) against the probability of requesting connections by each switch of the crossbar arbiter (all switches are assumed to have the same probability). We find that as the probability of switching requests is increased the improvement in performance is very gradual and approaches at most about 0.35. This means that each crossbar arbiter switch can make a connection at most about 35 % of the time even if it were able to request connections all of the time. This information is useful in predicting the performance of the arbiter for different "traffic" situations.

4.2 The Non-Synchronous RISC (NSR) Architecture

The NSR processor is a general purpose processor structured as a collection of self-timed units that operate concurrently and communicate over bundled data channels in the style of micropipelines [34]. The NSR pipeline stages correspond to standard synchronous pipeline stages such as Instruction Fetch, Instruction Decode, Execute, Memory Interface and Register File. However, each stage operates con-

currently as a separate self-timed process and are decoupled through self-timed FIFO queues between them that allow a high degree of overlap in their executions. Branches, jumps, and memory accesses are also decoupled through the use of additional FIFO queues which can hide the execution latency of these instructions. The prototype implementation of the NSR has been constructed using Actel Field Programmable Gate Arrays (FPGAs). In our Petri-net analysis, we include the Execute unit (EX), the Register file (RF), and the Instruction decode(ID). We will examine the performance of the EX independently and then its performance in association with the RF and ID stages.

Figure 6 shows the Petri Net representation of the EX unit of the NSR processor. The firing durations of the transitions (not shown in the diagram for the sake of clarity), were obtained from the circuit implementations of the EX unit. The flow of the Petri Net is as follows. First, the EX unit get the instruction from the ID unit and checks if this instruction is a *mvpc* instruction. This happens with an estimated probability of 0.1—a number which can be determined from execution traces. Once this is checked, if it is not a mvpc instruction, it is again checked whether it is a one operand or a two operand instruction. In case of a two operand instruction, the two operands are obtained sequentially from the RF (left-hand side shaded region in the Petri net) because of pin limitations. (The penalty of doing this will be analyzed momentarily.) This is followed by ALU operations where each instruction has a particular probability of occurrence. This portion of the Petri-net can be analyzed separately as shown by the legend labeled "*Refine*", and its evaluation time can be plugged back on the transition labeled `alureq`. Once the instruction is executed, the results are concurrently sent to the respective destinations based on the nature of the instruction.

To verify the correctness of our tools, we ran some examples on the Petri net and compared it against the performance of the actual implementation of the NSR For example, in the NSR chip, performance was measured by placing a series of 1000 instructions in a large loop and then executing the loop 65536 times [31]. The Petri net model was modified corresponding to each actual test to incorporate the probabilities of the instructions. For example, for the program with only the add instructions repeated 1000 times, the probabilities in the Petri net were modified to reflect this information by suitably changing the branching probabilities. The performance of the execute unit was found both analytically and through actual chip delay measurements for that instruction executing. It was found that the performance figures track the actual performance of the chip reasonably as shown in the graph 7. Statistical and modeling errors would account for much of the difference.

Once the tools were found to be reasonably correct, our first set of experiments were to estimate the performance of the EX unit with- and without considering pin limitations that sequentialize multiple

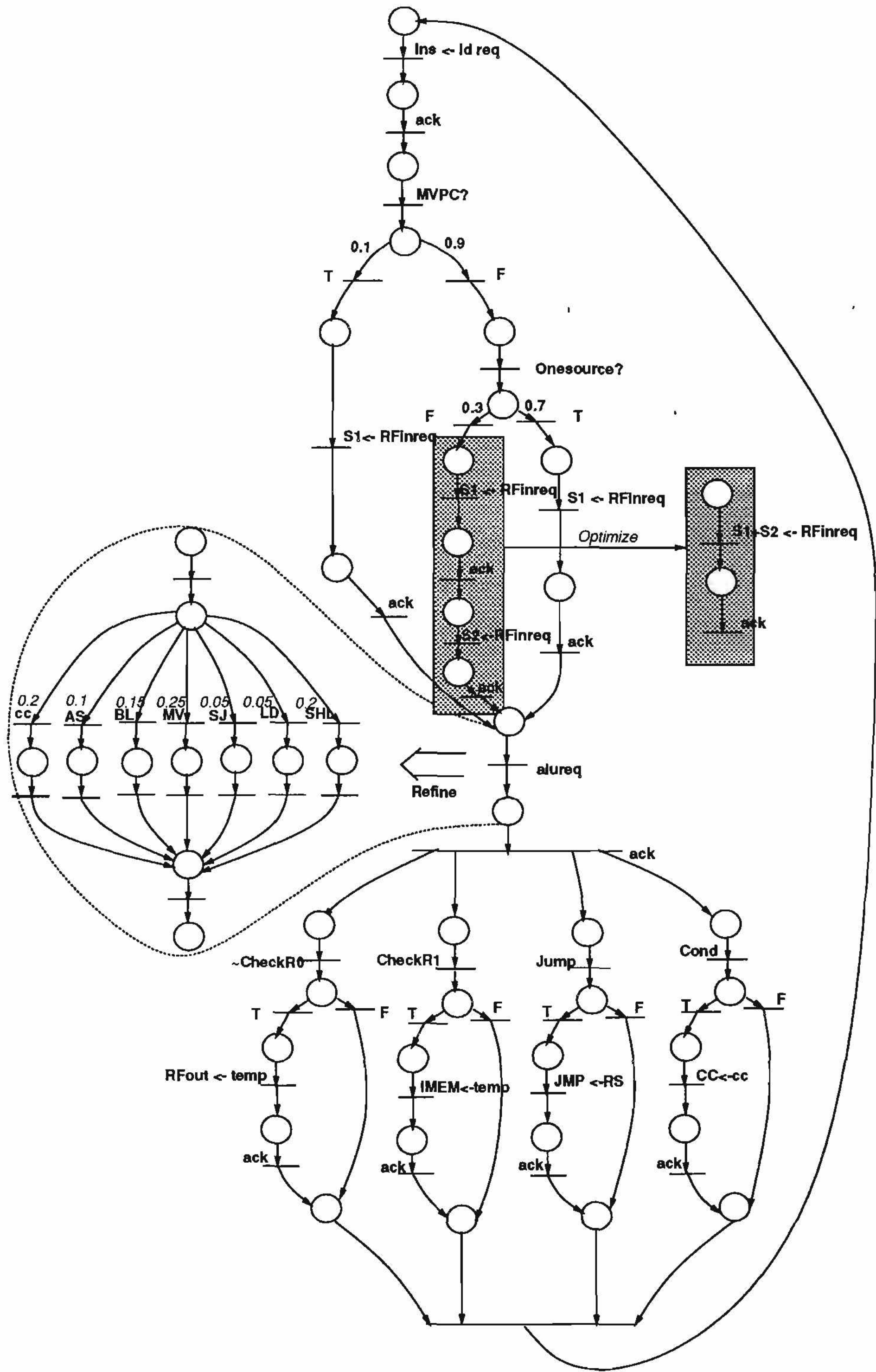

Figure 6: Petri Net Model of the Execute Unit

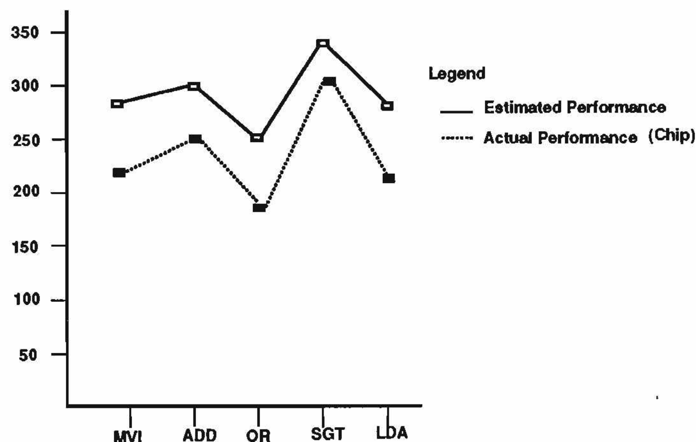

Figure 7: Comparison of Actual and Estimated Performance

operand access. In Figure 6 we have shown in the left-hand side shaded area the multiplexed version of the access and in the right-hand side shaded area the non-multiplexed access. In the analysis, we changed the portion of the Petri Net to effect the optimization. The estimated long run times of the two Petri Nets were compared. It was found that this Petri net has a single recurrence class in both cases. Comparing the long-run times for each module, we found that the latter design (non-multiplexed), effects a performance improvement of 5 % under the given module delay assumptions as well as branching probabilities.

Another problem that is immediately noticeable is that each time any operation executes in the execution unit, it has to be checked whether it is a MVPC instruction or not. We would like to compare, the effect this instruction has on the overall performance of the execute unit. The Petri net was modified to remove the *mvpc* instruction and the long run times of the two Petri nets (with and without mvpc) were compared. Through this analysis, we determined that the overhead of this is about 2 %. If this figure had been higher, then we would have performed further analysis such as the cost of having some alternate instruction which has the same effect as the mvpc and also we would have studied the effect on the performance of the processor as a whole.

Other possible analysis and estimates possible are briefly mentioned below. We can also estimate the effect of adding instructions in the EX unit, and then estimate the effects on the whole processor. It is encouraging that the GTPN approach can yield performance results that take into account module delays as well as branching probabilities and give answers to questions such as these in virtually interactive time. We can also make decision regarding whether to add buffers or the number of buffers to add in transferring the results from the EX unit to the RF unit, depending on the usage of these buffers.

Table 1: Table of experimental results

Circuit Name	Num of states	Performance (secs) (Long Run Time)	Analysis Time CPU (secs)
EXOPT	223	162	0.25
EX-RF	223	142	0.25
EX-MVPC	223	156	0.25
WAVE1	428	101	0.13
WAVE2	428	149	0.13
WAVE3	428	212	0.13
ARP	44	39	0.05
ARP-OPT	44	34	0.05

We can calculate the resource usages of these buffers in making the decision.

4.3 Summary of Examples

Table 1 shows the performance of various optimized versions of the NSR processor. First we studied the original execution unit (EXOPT), then compared it with the optimization of having a single 16 bit port to the RF (EX-RF) and then used it to calculate the performance without the mvpc instruction (EX-MVPC). The table also shows the estimated performance of some versions of the wavefront arbiter. The performances figures for different probabilities of firing of the crossbar switch are compared and some of them are shown as WAVE1, WAVE2 and WAVE3 in the table. An Asynchronous Reordering Pipeline [21] has also been analyzed. This circuit is a modified asynchronous micropipeline which implements a dynamic instruction reordering scheme that optimizes multiple pending instructions to achieve higher throughput. ARP is an unoptimized version of this circuit and ARP-OPT an optimized version based on data dependent probablities of the reordering requests.

5 Conclusions

Our experiments with the performance analysis methodology have been very encouraging. We have been able to analyze reasonably large examples and have been able to obtain the performance estimates in less than a second of CPU time in most cases as shown in Table 1. By this, we have provided a framework for comparing the performance of different asychronous circuits which have data dependent

behavior. We have demonstrated that the information obtained from analysis can be used in performance versus cost tradeoffs during the optimization process as well as studying the performance of a design under various constraints.

Although the GTPN analysis tools are very efficient in generating the reachability graph and use good numerical techniques for analysis, the problem of state explosion in generating the reachability graph of the Petri net can be significant in some cases. This was noticed while analyzing another version of the crossbar arbiter called the crisscrossing arbiter [14] which is very decoupled in its execution even though the circuit in itself is small. Generation of the reachability graph using symmetries, as well as verification of the correctness of circuits using symmetries has been discussed in [33] and [19] respectively; these techniques will be further studied.

Hierarchical reasoning can, in general, introduce errors in the analysis, because of the fact that the performance of the underlying net is abstractly represented by its long-run time. For the NSR processor's execution unit, we performed the analysis with- and without hierarchy, and the results were in close agreement. The exact magnitude of the error will be studied in our future work.

Another possible source of inaccuracy in our current approach is the following. In modeling the delay exhibited by data blocks whose computations are data dependent at a fine-grain level (*e.g.*, carry-completion addition), we currently take the *average delay* exhibited by those units with respect to the distribution of the data being handled by those units. A more direct way to incorporate the data dependent delays of data blocks has to be investigated.

Finally, macromodule based circuits lend themselves very well to Petri net based analysis. However, we would also like to consider other classes of asynchronous circuits (*e.g.*, burst-mode controllers [28]) as part of our future work.

To summarize, the main contribution of this paper has been to provide a framework for the analysis of asynchronous self-timed circuits taking into account data dependent behavior. This has been achieved by presenting techniques to represent any interconnection of self-timed blocks as a Petri Net and adapting techniques in the area of Petri net based performance analysis. As far as we know this is the first serious attempt in the area of Petri net based performance analysis of self-timed circuits. We have demonstrated our approach on reasonably large examples which explore different aspects of asynchronous circuit behavior.

References

1. AJMONE MARSAN, G. B., AND CONTE, G. *Performance Models of Multiprocessor Systems.* The

MIT Press, 1986.

2. Akella, V. *An Integrated Framework for the Automatic Synthesis of Efficient Self-Timed Circuits from Behavioral Specifications.* PhD thesis, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1992.

3. Akella, V., and Gopalakrishnan, G. SHILPA: A High-Level Synthesis System for Self-Timed Circuits. In *International Conference on Computer-aided Design, ICCAD 92* (Nov. 1992), pp. 587–591.

4. Brunvand, E. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, 1991.

5. Brunvand, E. The NSR processor. In *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume 1* (Jan. 1993), T. Mudge, V. Milutinovic, and L. Hunter, Eds.

6. Burns, S. Performance evaluation of asynchronous circuits. Tech. Rep. TR-91-1, Computer Science Dept., California Institute of Technology, 1991.

7. Camposano, R., and Wolf, W. *High Level VLSI Synthesis.* Kluwer Academic Publishers, 1991.

8. Chu, T.-A. *Synthesis of Self-timed VLSI Circuits from Graph Theoretic Specifications.* PhD thesis, Department of EECS, Massachusetts Institute of Technology, Sept. 1987.

9. Davis, A., Coates, B., and Stevens, K. The Post Office Experience: Designing a Large Asynchronous Chip. In *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume 1* (Jan. 1993), T. Mudge, V. Milutinovic, and L. Hunter, Eds., pp. 409–418.

10. Dill, D. L., Nowick, S. M., and Sproull, R. F. Specification and Automatic Verification of Self-Timed Queues. Tech. Rep. CSL-TR-89-387, Computer Systems Laboratory, Stanford University, Aug. 1989.

11. Ebergen, J. C. *Translating Programs into Delay Insensitive Circuits.* Centre for Mathematics and Computer Science, Amsterdam, 1989. *CWI Tract 56.*

12. E.Williams, T., and Horowitz, M. A zero-overhead self-timed 160ns 54bit cmos divider. *IEEE Journal of Solid State Circuits 26*, 11 (Nov. 1991), 1651–1661.

13. FURBER, S. Lectures on the Asynchronous ARM Processor given at the *VII Banff Workshop on "Asynchronous Hardware Design", Banff, Canada, August 28-September 3, 1993. (Organizer: Graham Birtwistle).*

14. GOPALAKRISHNAN, G. Some unusual micropipeline circuits. Tech. Rep. UUCS-93-015, University of Utah, Department of Computer Science, 1993.

15. HENNESSY, J., AND PATTERSON, D. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 1990.

16. HENRIK HULGAARD, STEVEN M BURNS, T. A., AND BORIELLO, G. An algorithm for exact bounds on the time separation of events in concurrent systems. Tech. Rep. UW-CSE-94-02-02, Department of Computer Science, University of Washington, 1994.

17. HOLLIDAY, M., AND VERNON, M. The GTPN analyzer: Numerical methods and user interface. *IEEE Comp Soc. 1986 Call Joint Computer Conference* (Nov. 1986).

18. HOLLIDAY, M., AND VERNON, M. A generalized timed petri net model for performance analysis. *IEEE Transactions on Software Engineering 13* (Dec. 1987), 1297–1310.

19. IP, C. N., AND DILL, D. Efficient verification of symmetric concurrent systems. In *Proceedings of the International Conference on Computer Design (ICCD)* (1993), pp. 230–234.

20. LEON-GARCIA, A. *Probability and Random Processes for Electrical Engineering.* Addison Wesley Publishing Company, 1989.

21. LIEBCHEN, A., AND GOPALAKRISHNAN, G. Dynamic reordering of high latency transactions in time-warp simulation using a modified micropipeline. In *International Conference on Computer Design (ICCD), 1992* (1992). Accepted for publication. Also available as Technical Report UUCS-TR-92-005, Department of Computer Science, University of Utah.

22. MEAD, C. A., AND CONWAY, L. *An Introduction to VLSI Systems.* Addison Wesley, 1980. Chapter 7 entitled "System Timing".

23. MISUNAS, D. Petri nets and speed independent design. *Communications of the ACM 16*, 8 (Aug. 1973), 474–481.

24. MOLLOY, M. K. Performance analysis using stochastic petri nets. *IEEE Transactions on Computers C-31* (1982), 417–423.

25. MURATA, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* (1989).

26. NIELSON, C. D., AND KISHINEVSKY, M. Performance analysis based on timing simulation. Tech. Rep. ID-TR:1993-125, Department of Computer Science, Technical University of Denmark, 1993.

27. NOWICK, S., DEAN, M., DILL, D., AND HOROWITZ, M. The design of a high-performance cache controller: A case study in asynchronous synthesis. In *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume 1* (Jan. 1993), T. Mudge, V. Milutinovic, and L. Hunter, Eds., pp. 419–427.

28. NOWICK, S. M., YUN, K. Y., AND DILL, D. L. Practical asynchronous controller design. In *Proceedings of the International Conference on Computer Design* (1992 (to appear)).

29. PENG, Z., AND KUCHCINSKI, K. Automated transformation of algorithms into register-transfer level implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 2 (Feb. 1994), 150–166.

30. RAMAMOORTHY, C., AND HO, G. Performance evaluation of asynchronous concurrent systems using petri nets. *IEEE Transactions on Software Engineering SE-6*, 5 (Sept. 1980), 440–449.

31. RICHARDSON, W. F., AND BRUNVAND, E. The NSR Processor Prototype. Tech. Rep. UUCS-92-022, Department of Computer Science, University of Utah, Oct. 1992.

32. SPARSOE, J., AND STAUNSTRUP, J. Design and performance evaluation of delay insensitive multi-ring structures. In *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume 1* (Jan. 1993), T. Mudge, V. Milutinovic, and L. Hunter, Eds.

33. STARKE, P. H. Reachability analysis of petri nets using symmetries. *Systems Analysis - Modeling - Simulation 8*, 4/5 (1991).

34. SUTHERLAND, I. Micropipelines. *Communications of the ACM* (June 1989). *The 1988 ACM Turing Award Lecture.*

35. TAMIR, Y., AND CHI, H.-C. Symmetric crossbar arbiters for VLSI communication switches. *IEEE Transactions on Paralled and Distributed Systems 4*, 1 (1993), 13–27.

36. WILLIAMS, T. E. *Self-Timed Rings and Their Applications to Division.* PhD thesis, Department of Computer Science, Stanford University, May 1991.

37. ZUBEREK, W. Timed petri nets and preliminary performance evaluation. In *7th Annual International Symposium on Computer Architecture* (1980), pp. 88–96.