Formal Approaches to Globally Asynchronous and Locally Synchronous Design

Bin Xue

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

August 22, 2011
Blacksburg, Virginia

Keywords :marked graph, latency insensitive system, scheduling, back pressure, throughput, polychronous formalism, verification.

Formal Approaches to Globally Asynchronous and Locally Synchronous Design

Bin Xue

(ABSTRACT)

The research reported in this dissertation is motivated by two trends in the system-on-chip (SoC) design industry. First, due to the incessant technology scaling, the interconnect delays are getting larger compared to gate delays, leading to multi-cycle delays in communication between functional blocks on the chip, which makes implementing a synchronous global clock difficult, and power consuming. As a result, globally asynchronous and locally synchronous (GALS) designs have been proposed for future SoCs. Second, due to time-to-market pressure, and productivity gain, intellectual property (IP) block reuse is a rising trend in SoC design industry. Predesigned IPs may already be optimized and verified for timing for certain clock frequency, and hence when used in an SoC, GALS offers a good solution that avoids reoptimizing or redesigning the existing IPs. A special case of GALS, known as Latency-Insensitive Protocol (LIP) lets designers adopt the well-understood and developed design flow of synchronous design while solving the multi-cycle latency at the interconnects. The communication fabrics for LIP are synchronous pipelines with hand shaking. However, handshake based protocol has complex control logics and the unnecessary handshake brings down the system's throughput. That is why scheduling based LIP was proposed to avoid the hand-shakes by pre-calculated clock gating sequences for each block. It is shown to have better throughput and easier to implement. Unfortunately, static scheduling only exists for bounded systems. Therefore, this type of design in literatures restrict their discussions to systems whose graphic representation has a single strongly connected component (SCC), which by the theory is bounded.

This dissertation provides an optimization design flow for LIP synthesis with respect to back pressure, throughput and buffer sizes. This is based on extending the scheduled LIP with minimum modifications to render it general enough to be applicable to most systems, especially those with multiple SCCs. In order to guarantee the design correctness, a formal framework that can analyze concurrency and prevent fallacious behaviors such as overflow, deadlock etc., is required. Among many formal models of concurrency used previously in asynchronous system design, marked graphs, periodic clock calculus and polychrony are chosen for the purpose of modeling, analyzing and verifying in this work. Polychrony, originally developed for embedded software modeling and synthesis, is able to specify multi-rate interfaces. Then a synchronous composition can be analyzed to avoid incompatibly and combinational loops which causes incorrect GALS distribution. The marked graph model is a good candidate to represent the interconnection network which is quite suitable for modeling the communication and synchronizations in LIP. The periodic clock calculus is useful in analyzing clock gating sequences because periodic clock calculus easily captures data dependencies, throughput constraints as well as buffer sizes required for synchronization. These formal methods help establish a formally based design flow for creating a synchronous design and then transforming it into a GALS implementation either using LIP or in a more general GALS mechanisms.

This project received partial financial support from NSF grant CCF-0830882 and SRC task 1818.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Globally Asynchronous and Locally Synchronous (GALS) systems are likely to dominate the future integrated circuits (ICs) that are designed as system-on-chips (SOC). This seems like an inevitable consequence of the Moore's law [47], combined with the increasing intellectual property (IP) reuse when creating an SOC, and the need for low-power system design. In this dissertation we consider a number of problems germane to GALS design. The problems facing the IC industry that gave rise to this increasing interest in GALS research are as follows: the first problem arises from the unceasing improvement in semiconductor manufacturing technology. The feature sizes are decreasing all the time which allows increasing number of devices fitted on a single die. However, this inevitably makes it more difficult to design a global-clock network that can control all the blocks in the design. Even when this can be done, such a network significantly increases the power consumption. Secondly, the SOC design methodology requires reusing predesigned and verified IP modules to shorten the design to market cycle by enabling the designers to put more concentration on modification and customization at system level. However, the speed of each IP module is often designed and optimized for different clocks by different groups. Global optimization is hard to achieve without changing the predesigned IP blocks. Thirdly, for certain clan of designs, multiple clock domains are inherently required according to the nature of their functionalities. This may be due to the fact that they may perform different computations and communications. Partitioning large system into several clock domains is a possible solution to these problems mentioned above, which forms a GALS system. More detailed review for related work on GALS design styles is provided in Section 3.1.

1.1 Latency Insensitive Design

Latency Insensitive Design (LID) [16], some times also regarded as *elastic design*, is a special case of GALS design that it is still a synchronous design but adopts the idea of hand-shake to control the data flows as used in asynchronous design. Remaining in synchronous domain allows the LID to utilize the prevailing design flows of synchronous design with only small modifications. LID provides a flexible assembling of IP blocks, making them *elastic* to interconnect latencies. This approach packs IP blocks within *wrappers* and inserts *relay stations* (RS), a special communication pipeline between IP blocks according to the post layout latency information. It can compensate for wiring latencies and synchronize data flows with different speeds. The resulting system is regarded as a Latency Insensitive System (LIS) (shown in Fig. 1.1(a)). In that figure, RS1, RS2, RS3 and RS4 are four relay stations transporting two data flows f1 and f2 to an IP module, which is packed within a wrapper.

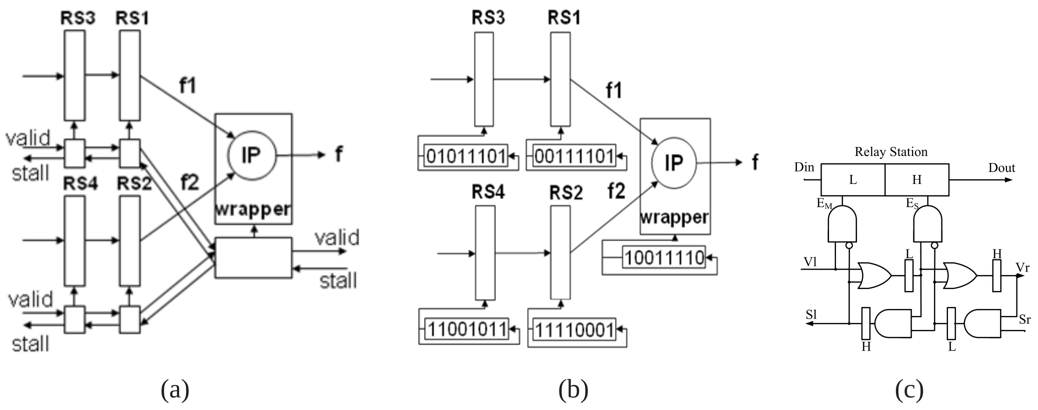


Figure 1.1: (a) A hand-shaking based LIS (b) LIS by scheduling (c) Implementation of EB

An important problem in this context is how to control such interconnect pipelines. Early work on LIS design such as [33, 42] use *hand-shake based protocols*. Every stage of the pipeline is controlled by *valid* and *stall* signals, where "valid=1" denotes that the current the input from its previous pipeline stage is valid, "stall=1" denotes that the next stage of the pipeline is full and the current stage needs to be stalled from generating new outputs. The use of both valid and stall signals indicating that the execution of a stage in the pipeline depends both on the availability of valid inputs and the availability of space at the output buffers of this stage. Stall signal is generated and transmitted in the opposite direction of the data flow and it can disable its previous stage from execution when the receiving stage has no available storage space. This way of using stall signal is also termed as "back pressure" in the literature [17]. The link sending back pressure information is referred to as "back pressure arc" accordingly. More detailed explanation about back pressure can be found in Section 2.1. A common implementation of the pipeline with hand-shaking is the *elastic buffer* (EB) [33] shown in Fig. 1.1(c). The clock signal is not shown for the sake of simplicity. The rectangular structures marked "H" and "L" are latches which are enabled when clock value is "1" and "0" respectively. Its data path consists of two latches clock gated by "E_M" and "E_S" which depend on the hand-shaking signals. Such a structure allows storing two data values when the pipeline is stalled. Similar implementations include the *phase elastic buffer* (pEB) [80] which is claimed to be compatible for both synchronous and asynchronous interfaces, and a robust wrapper [2] which can endure multi-clock cycle latencies. However, the control logic for these hand-shake based protocols is complicated which results in large area overhead.

If the clock gating signals are periodic, the control logic in Fig. 1.1(a) can be replaced by periodic clock gating sequences (schedulings) leading to the system shown in Fig. 1.1(b). This alternative approach is called a *scheduling based protocol* [20]. It avoids the *stalling signals* by clock gating the relay stations and wrappers with pre-calculated scheduling sequences which reduces the area overhead incurred by the complicated handshake controlling blocks. Hence, it is suitable for applications where area is critical. Another parallel attempt is presented in [12]. In this approach, they use an as soon as possible (ASAP) scheduling rule so that an IP module is activated as soon as all of its inputs are available. A more rigorous discussion about ASAP scheduling is provided in Section 2.1. On the other hand, ASAP scheduling is a scheduling for unconstrained output buffer size and data may aggregate at a reconvergent IP module (a module which has multiple input channels) because of resynchronization. It requires additional buffers to store these data and at the same time not affect the system's performance [1]. Therefore, implementations of *fractional synchronizers* [20] and *fractional registers* [12] are used to *equalize* the system since every *elementary cycle* (see Definition 3.4) of the design must have the same throughput.

1.2 Modeling and Analyzing LIS Implementation Using Polychronous Framework

Implementation of GALS calls for innovate verification methods, since the verification tools for synchronous design can't be directly applied to asynchronous systems. However, only a few verification tools have been proven useful for asynchronous and GALS designs but there are some attempts. A review of these works is provided in Section 3.2.4. Most of the current work are based on model checking techniques which verify the protocol by exploring all of the reachable states. This is not desirable for GALS design. The GALS design has asynchronous communication and exhibits indeterministic behavior. Multiple execution path can exist leading to more observable states. Therefore, the state explosion problem is worse than in completely synchronous designs. It calls for a new approach other than model checking based technique for the purpose of analysis and verification.

On the other hand, the distributed embedded software community has been working on GALS in a slightly different setup for at least two decades [7] [6][9]. For example, a synchronous language SIGNAL [4] has been developed to allow people to describe their designs with concurrency specification and to concentrate on the functional correctness without considering the timing issue arising in the real implementation, especially for a distributed environment where the concept of GALS applies. The task of generating and verifying an implementation independent from communication delay, is handled by the SIGNAL compiler Polychrony and a verification tool SIGNALI. In [7][6], Benveniste et. al. formally model synchronous, asynchronous processes, desynchronization, resynchronization and composition of processes under synchronous or asynchronous environments. Based on these works, formal definition of endochrony and isochrony [7] are proposed.

[1]An detailed explanation can be found in Section 5.2.1

Endochrony refers to the property of a process which enforces unique scheduling and deterministic interaction with its asynchronous environment. Isochrony refers to a delay-independent communication between a pair of processes. Endochrony and isochrony together forms endo-isochrony which is a sufficient property for correct GALS implementation [7][6] for distributed embedded software regardless of communication delays. A. Benveniste et al. [73] formalize the functionality of LIPs and transform the synchronous modules of a multi-clock synchronous specification into weakly endochronous modules for which simple and efficient wrappers exists. In [72], Talpin unifies the verification of endochrony, isochrony and weakly endochrony by checking the formal structures Synchronous Data Flow Graph (SDFG) [58] and Clock Hierarchy (CH) [3]. [62] proposes *synchronous structure* to describe polychrony from an event based perspective, which provides a more direct view of the semantics in SIGNAL. The theory concerning GALS combined with the formal tools provide a possible solution of checking correct communication for a given system. However, most of these works are usually hired in the embedded software domain. To the best of our knowledge, not much work has been done for formalizing hardware GALS implementations, especially for the designs described at the RTL or gate level.

1.2.1 Contribution towards modeling and analyzing LIS implementation using polychronous framework

We present a framework of modeling and analyzing the communication for LIPs based on the Polychrony framework. The Polychrony framework consists of the SIGNAL language and various analysis tools and software synthesis tools. The static analysis of various properties of a SIGNAL program are done using formal structures of SDFG and CH . In our framework for analysis of LIP based system, a given LIP is firstly modeled by the SIGNAL syntax which is further transformed to obtain its SDFG and CH. Next, endo-isochrony is checked based on the obtained SDFG and CH and their respective composition. A case study of modeling and analyzing Phase Elastic Buffer (pEB) is done to demonstrate the verification flow and validate the proposed framework. The prominent distinction between our framework and other verification techniques for GALS is that we are using static analysis which is an efficient technique. Also, we bring the concept of Polychrony to real hardware level LIP based SOC communication. However, we must point out that endo-isochrony is an over restriction, some systems which don't satisfy this property can still be implemented with correct functionality.

1.3 Synthesis of Scheduled LIP

Handshake based protocol which uses "stall" signals (back pressure) for every communication link has several disadvantages.

1. Handling back pressure requires relatively complicated logics thus using back pressure for

every communication link introduces large overhead.

2. Each pipeline stage has a buffer size of two may causes unnecessary stalls which brings down the system's throughput.

3. Techniques [56, 15] have been proposed to improve the system's throughput with buffer resizing or buffer addition. They rely on solving a MILP formulation which is NP-hard. Particularly, the integer variable numbers of this MILP formulation equals back pressure numbers which does not scale for big systems.

Scheduled LIP seems be a good alternative to overcome these disadvantages. However, the current research about scheduled LIP also has limitations.

1. Research of scheduled LIP is limited to a system whose graphic representation has only one strongly connected component (SCC). For a LIS with mulitple-SCC, completely removing handshake may cause overflow thus unable to get a static scheduling for the system.

2. Implementation of this protocol has not been fully studied. Current research simply uses shift registers to store the scheduling sequence of one block which are not area efficient. It may cause even more area if the scheduling sequence is long.

Therefore, we propose a novel flow for LIS optimization. This flow extends the analysis for multiple SCC system, given that most current works only consider systems with one SCC.

The proposed design flow for any LIS is shown in Fig. 1.2. It starts from the floor planning before which the design flow of scheduled LIS is the same as normal synchronous digital design. Relay stations and Wrappers are added to the original netlist to get netlist* based on the latency information obtained during the floor planning and routing. Netlist* is an incomplete net list before clock gating sequences are implemented. Then we propose *partially back pressure graph* (PBPG), an extension of *marked graph* (MG) (the background of MG is provided in Section 2.1), to model the system, this is according to Process A shown in Fig. 1.2. At this moment, no back pressure is used. Then we determine whether overflow exists for this PBPG model, since up to now we don't have any hand-shakes in the system and should conservatively assume that overflow might happen. This directs the design flow into process B shown in Fig. 1.2. If this model is not bounded, some modifications are required to make it bounded by ASAP scheduling, which relates to the process C shown in Fig. 1.2. Now we can be confident that the resulted system is bounded. Due to the side effect of back pressure, throughput might drop. If a throughput reduction is observed, Process D increases system's throughput to its maximal achievable value. In process E, schedulings constraints are represented and analyzed. Such schedulings should take data dependency and throughput restriction into consideration. After this, we need to allocate some additional buffer size for synchronization. At reconvergent blocks that have multiple inputs, data may get aggregated because of barrier synchronization. The buffer sizes at these locations should be computed. This is done in Process E. In the end, the control logics for generating the

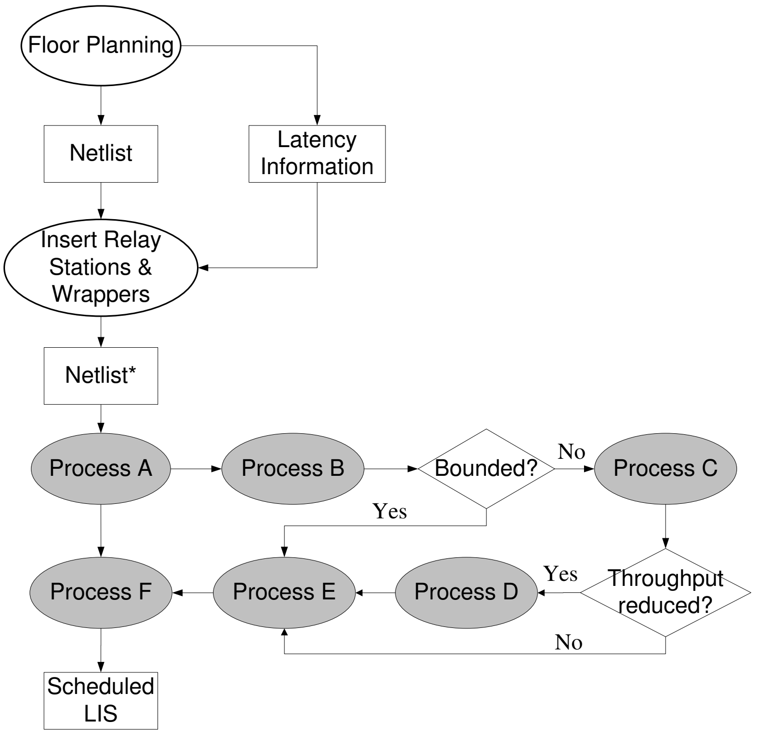


Figure 1.2: Synthesis flow of scheduled LIS

schedulings as well as the barrier synchronization and back pressure are implemented in Process F which completes the synthesis flow.

Let's take a close look at these processes mentioned in the synthesis flow which are not intuitive. They are related to the problems we need to solve before establishing the design flow.

1. Process A models an LIS with or without back pressure. Our optimization for LIS relies on using handshake (back pressure) for some of the communication links. We need a model which is able to represent this scenario. Performance of an LIS such as throughput and buffer sizes should be conveniently expressed.

2. Process B determines whether the system is bounded. By the existing research in Petri-nets, if a system is strongly connected i.e. its graphic representation has only one *strongly connected component*(SCC), the system is bounded for any scheduling sequences including the ASAP scheduling. That is why all of the previous work on scheduled LIS assume the system is strongly connected. However, this is not the case for most systems. Whether a system with more than one SCC is bounded is not clear. It calls for a practical approach to make this decision.

3. Process C modifies the system if it is evaluated to be unbounded in process B. In this case, hand-shakes should be employed to *balance* the different data flows. Obviously, we don't

want to use hand-shake for every communication channel, otherwise the LIS becomes the hand-shake based and possibly highly non optimal for throughput. So it should be used in a smart way so that only necessary communication links are implemented with back pressure to make the system bounded. However, no previous work has formally addressed this issue based on our knowledge.

4. Process D improves the LIS's throughput if a reduction is observed after Process C because of newly formed critical cycles. Techniques [56, 15] have been proposed to improve the system's throughput with buffer resizing or buffer addition. They rely on solving a MILP formulation which is NP-hard. However, their methods are suitable for a LIS where back pressure are used for every communication link not PBPG. Also, these techniques do not scale for big systems.

5. Process E analyzes and computes the schedules for each block, also computes the number of fractional synchronizer used at the reconvergent block. Schedule (clock gating) sequences represent whether blocks are executed or disabled in the current clock cycle and are periodic binary sequences consisting "1" and "0"s. "1" represents that the current block is activated while "0" represents not. It's favorable to treat each schedule sequence as a variable and derive the dependencies and throughput constraints in terms of these scheduling variables. Based on this, optimizations related to scheduling, e.g. buffer sizes, can be performed. If the scheduling sequences are obtained, a convenient approach should be provided to get this information. Another interesting problem is whether this can be done or estimated with accuracy before the schedules are computed.

6. Process F implements the schedule (clock gating sequences) but the following issues may arise in the implementation of the scheduling sequences. Generating the scheduling sequences for every block from a centralized module will cause routing problems. On the other hand, storing the clock gating sequences in local shift registers may lead to more area overhead than it saves, especially when the period of the clock gating sequence is long. Therefore, it requires an area efficient implementation for the ASAP scheduling.

Parts of the dissertation's contributions were solving the problems highlighted in the six processes. Process E mainly deals with the binary sequences. The synchronous languages community has been studying the characteristics of such periodic scheduling sequences, and they call each such sequence a periodic clock. In [29, 28, 30], a framework called *periodic clock calculus* is used to analyze the composition of clocks when operations such as *down sampling*, *delay*, *resynchronization* etc., are applied. This calculus also allows them to synthesize buffers of appropriate sizes in order to synchronize events from different clock cycles. We find that their framework is also quite appropriate for these two processes in the design flow of scheduled LIS. On the other hand, Process A, Process B, Process C and Process D are relatively difficult. Analytical model should be created and it requires rigorous formal proofs along the way each derivation is made.

1.3.1 Contribution towards analysis of LIS using periodic clock calculus

We apply the periodic clock calculus for the issues involved in processes E mentioned above.

1. We model an LIS by a marked graph [61] using the modeling technique described in Section 2.1.3. We relate the scheduling sequences of different transitions within the clock equations and use a *subtyping relation* (see the definition in Section 2.2) to infer the throughput constraints. We show that our method gives a result compatible to [14]. In addition, we demonstrate the fact that our method can derive all the scheduling constraints.

2. We employ ideas from the periodic clock calculus to infer the number of fractional synchronizers used at each *join transition* (the reconvergent transition with multiple inputs). Also, the activating sequence (scheduling sequence) of each fractional synchronizer is determined. We prove that our way of synthesizing the fractional synchronizer is sufficient to make the system's marked graph 1-safe (each place has at most one token for all time). The importance of making the system 1-safe will be discussed in Section 5.2.2

3. We propose a static method to estimate the number of fractional synchronizers for a given elementary cycle based only on the structure of the system. With this method, one can grab an earlier estimate of the fractional synchronizers before the complex computation for the system's schedules.

1.3.2 Contribution towards optimization of LIS based on back pressure minimization

Here we summarize our effort towards Process A, B, C, D and F mentioned before. We model the LIS by PBPG, which is used for all processes in the design flow. PBPG models the change of an LIS when back pressure arcs are gradually added. We start our exploration for system boundness by observing that the back pressure must be used to avoid overflow when a faster SCC sends data to a slower SCC. We extend this observation to multiple SCC interconnection situation and formally show a necessary and sufficient condition to decide whether the boundedness is satisfied. The problem of finding a minimal set of stalling signals was formalized as the Minimization of Back Pressure Arcs (MBPA) problem, which can be solved efficiently by a transformation to the Minimum Cost Arborescence (MCA) problem. After MBPA, back pressure arcs may form new critical cycles, which could possibly bring down the resulted system's throughput. If this happens, we provide two techniques *reduced integer mixed linear programming* (RMILP) and *localized integer mixed linear programming* (LMILP) to increase the LIS's throughput. Meanwhile, we provide a decentralized implementation for generating scheduling sequences which is expected to have fewer area cost. To the best of our knowledge, this work is the first one that bridges the gap between the hand-shake based LIS and the scheduled LIS. Our contributions toward the synthesis flow of scheduled LIS are summarized below:

1. We proposed PBPG, derived from marked graph, to model LIS. Initially, LIS has no back pressure. Then, back pressure arcs are gradually added to make the LIS bounded. PBPG is able to model this modification and is not only used in process A but used throughout the design flow.

2. We propose the notion of a Strongly Connected Component Graph (SCCG) and based on which a theorem is derived to determine whether such LIS is bounded.

3. We provided an assumption based on which algorithm (MBPA) is proposed to searches for the minimal back pressures required to prevent overflow. MBPA is shown to be solvable as the well known *Minimum Cost Arborescences* problem which has efficient solution. This algorithm is applicable to general systems including those with multiple SCCs.

4. We propose RMILP and LMILP to improve the throughput of the system with minimal buffer size increase. We compare the RMILP and LMILP's scalability against the current methods, which demonstrate better performance.

5. With both the buffer size and handshake information, an LIS implementation is proposed. The design of relay stations, wrappers and back pressure is given. We also discuss about the the system's initialization such that the implementation refines the specification of its PBPG model.

1.4 Publications

The publications regarding this dissertation are listed below:

- Bin Xue, Sandeep K. Shukla, S.S. Ravi, Optimization of Latency Insensitive System based on Back Pressure Minimization, under reviewed for publication.

- Bin Xue, Sandeep K. Shukla. Optimization of back pressure and throughput for latency insensitive systems. In Proceedings of ICCD'2010. pp.45-51

- Bin Xue, Sandeep K. Shukla, S. S. Ravi. Minimizing back pressure for latency insensitive system synthesis. In Proceedings of MEMOCODE'2010. pp.189-198

- Xue, B. and Shukla, S. K. 2010. Analysis of Scheduled Latency Insensitive Systems with Periodic Clock Calculus. Journal of Electronic Testing. Volume 26, Number 2, April, 2010, Pages 227-242.

- Bin Xue, Sandeep K. Shukla, Analysis of Latency Insensitive Protocols Using Periodic Clock Calculus. Appears in IEEE International High Level Design Validation and Test Workshop (HLDVT'09), Nov. 2009, San Francisco, USA.

- Bin Xue, Sandeep K. Shukla, Modeling and Analysis of Latency Insensitive Protocols Using Signal Framework. In Proceedings of the 4th International Workshop on the Application of Formal Methods for Globally Asynchronous and Locally Synchronous Design (FMGALS 2009), April, 2009, Nice, France. Also appears in Electronic Notes in Theoretical Computer Science (ENTCS), Volume 245, 2 August 2009, Pages 3-22.
- Bijoy A. Jose, Bin Xue, Sandeep K. Shukla, An Analysis of the Composition of Synchronous Systems. In Proceedings of the 4th International Workshop on the Application of Formal Methods for Globally Asynchronous and Locally Synchronous Design (FMGALS 2009), April, 2009, Nice,France. Also appears in Electronic Notes in Theoretical Computer Science (ENTCS), Volume 245, 2 August 2009, Pages 69-84.

1.5 Dissertation Organization

This dissertation is organized according to the areas according to Fig. 1.3.

1. Chapter 2 introduces the fundamentals of marked graph's definition, behavioral properties and their use in modeling for LIS. Meanwhile, the basic notations of periodic clock calculus is mentioned. Then a brief description of polychronous framework is described. In addition, our tool LIS-LOLA is introduced.
2. Chapter 3 discusses the related work on GALS design and different formal approaches applied for GALS and LIS design analysis. Then it narrows down the scope to related work on elastic designs and talks about the existing model, and optimization for LIS.
3. Chapter 4 presents a framework which makes use of the polychrony tool set and associated semantic analysis techniques, in the form of endo-isochrony. We show a number of LIP protocols to preserve the correctness with respect to their fully synchronous specifications using our framework.
4. Chapter 5 shows the application of periodic clock calculus for analysis of schedule LIS. We demonstrate that the schedule constraints can be expressed by a group clock equations. We also show how the buffer size can be computed based on schedules. In addition, a buffer size estimation technique is given only by the net's structures(without computing the scheduling sequences).
5. Chapter 6 extends the ASAP scheduling for LIS whose graph representations have multiple SCCs. We show that back pressure are required for some inter-SCC arcs to make the system bounded by giving a sufficient and necessary condition. Based on that, an algorithm is derived which determines the minimal number of back pressure arcs required to make the system bounded with ASAP scheduling.

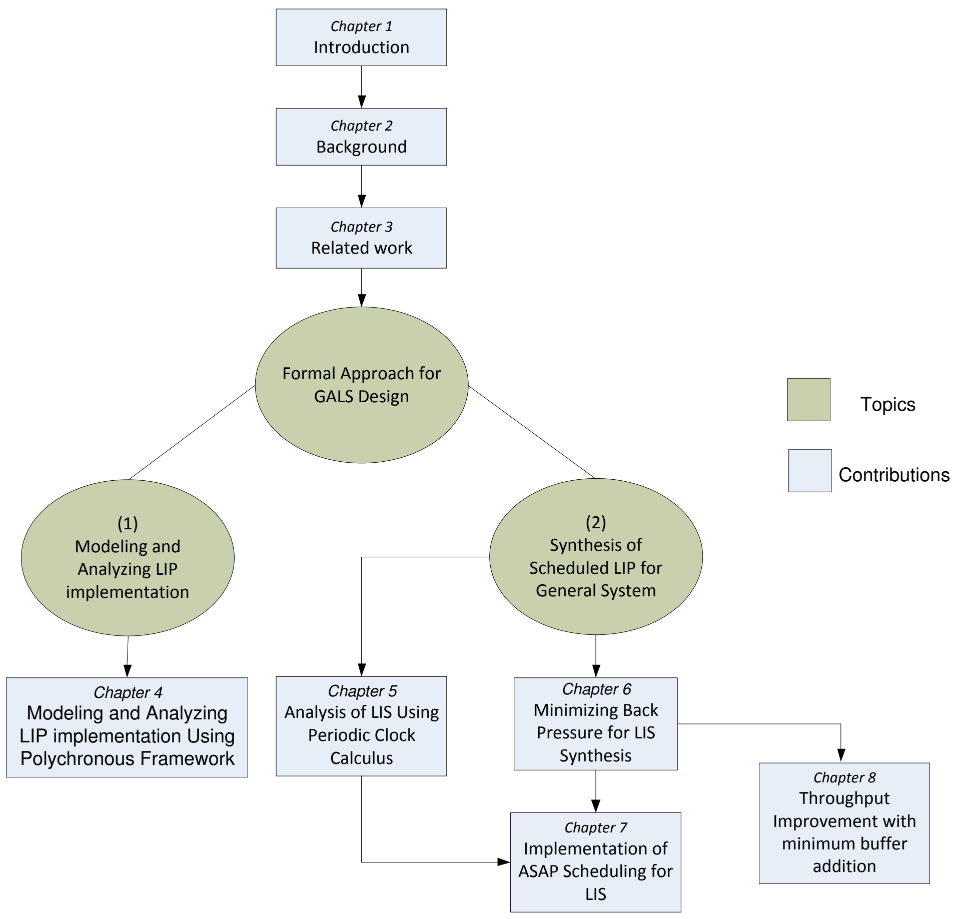

Chapter 1
Introduction
Chapter 2
Background
Chapter 3
Related work
Formal Approach for GALS Design
Topics
Contributions
(1) Modeling and Analyzing LIP implementation
(2) Synthesis of Scheduled LIP for General System
Chapter 4
Modeling and Analyzing LIP implementation Using Polychronous Framework
Chapter 5
Analysis of LIS Using Periodic Clock Calculus
Chapter 6
Minimizing Back Pressure for LIS Synthesis
Chapter 7
Implementation of ASAP Scheduling for LIS
Chapter 8
Throughput Improvement with minimum buffer addition

Figure 1.3: Organization of the dissertation

6. Chapter 7 discusses an implementation of ASAP scheduling for LIS. Instead of using a centralized scheduler to generate the scheduling sequences or shift-register to store a scheduling for each transition. Our implementation realizes the control in a distributed and area efficient manner.

7. Chapter 8 talks about throughput improvement. We are propose two mixed integer linear programming formulations for throughput improvement, which aims at handling the throughput degradation problem caused by back pressure arcs. Solving MILP formulation is known to be NP hard, so we are trying to get a MILP formulation with reduced size based on SCC level abstraction (RMILP) or localization (LMILP) Combined with the MBPA, the flow which can optimize the LIS's back pressure arc as well as the throughput is completed.

Chapter 2

Background

In this chapter, we introduce the basic concepts and definitions required for further discussion of our work. Marked graph (MG) is a formal modeling notation that we have used to represent the interconnections and communications of an LIS. In Section 2.1, a rigorous definition of MG is explained and its behavior properties are introduced. Then we employ the ASAP scheduling [12] for an MG to make it suitable for analyzing the state reachability of an LIS. We show that an LIS with strongly connected graph representation has periodic state transitions by ASAP scheduling. Based on this, one modeling technique to model an LIS with an MG is described. In Section 2.2, a periodic clock calculus is introduced. The definition of clock, periodic clock, clock synchronization are discussed. Some formal notations are briefly explained. In Section 2.3, a framework of polychrony [38] which we use to specify a multi-rate interface is discussed. A validation technique to check correct communication based on Endo/Isochrony is provided there after. In the end, we briefly introduce LIS-LOLA, the tool built based on LOLA [67] to validate our algorithms throughout the dissertation.

2.1 Marked Graph Model

2.1.1 Transition, Place, Marking and Firing rules

LIS is very well modeled by Marked Graphs (MG) [32], which is a subclass of *Petri-net*. In [61], one can find an excellent review of the basics. Our notations are based on [61].

Definition 1 (Marked Graph (MG))**.** *A marked graph is a Petri net (P, T, W, M_0) where P is a set of places, T is a set of transitions, W is a set of arcs such that $W \subseteq (P \times T) \cup (T \times P)$, M is the markings ($M : P \to \mathbb{N}$) with the restriction that every place P has exactly one incoming arc, and one outgoing arc. M_0 is used for the initial markings.*

According to the definition of Petri-net. An MG is a directed, bipartite graph whose nodes can be classified into two categories, the *transitions* and the *places*. Generally, the transitions are drawn as rectangles or bars and places are drawn as circles in the graphical representation. One directed arc connects a transition and a place. For MG, each place can have only one incoming and one outgoing arc. A marking M assigns each place p a nonnegative integer number $M(p)$ and it is said that the place p has $M(p)$ tokens. $M(p)$ tokens are drawn as $M(p)$ dots in p. A marking M relates to a state of the system being modeled by the marked graph and M_0 is the initial marking which relates to the initial state. Fig. 2.1(a) shows a simple MG model that consists of four transitions $T0$, $T1$, $T2$ and $T3$ and three places $P0$, $P1$ and $P2$. $M(P0) = 1$, $M(P1) = 0$ and $M(P2) = 1$.

An MG is generally used to model a concurrent system. Transitions are usually employed to model events in the concurrent system, the *firings* of transitions represent the triggering or execution of these events. Sometimes, places represent the conditions. In these cases, places connected to the incoming and outgoing arcs of a transition can be interpreted as the preconditions or the postconditions of firing for this transition. Nonzero markings of these places indicates that these conditions are true. In other cases, places can represent data transfer and they are interpreted as the input and output data of the transitions.

The following notation is usually used to express pre-set and post-set for each transition t and place p.

1. $^\bullet t = \{p|(p,t) \in W\} =$ the set of input places of t
2. $t^\bullet = \{p|(t,p) \in W\} =$ the set of output places of t
3. $^\bullet p = \{t|(t,p) \in W\} =$ the set of input transitions of p
4. $p^\bullet = \{t|(p,t) \in W\} =$ the set of output transitions of p

A transition t is enabled at marking M if $M(p) > 0$ for every $p \in {}^\bullet t$. In Fig. 2.1(a), $T1$ and $T3$ are enabled while $T2$ is not. However, a transition which is enabled may not be fired. Before any firing, the list of transitions which are enabled is called the enabled transition list. Each time a transition is selected from the enabled transition list to fire and firing of t leads to a new marking M' according to:

$$M'(p) = \begin{cases} M(p) - 1 & if\ p \in {}^\bullet t \backslash t^\bullet \\ M(p) + 1 & if\ p \in t^\bullet \backslash {}^\bullet t \\ M(p) & otherwise \end{cases} \tag{2.1}$$

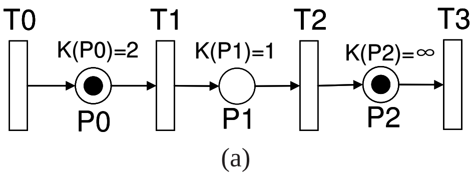
Such firing mechanism is perfectly suitable for modeling asynchronous behavior in a interleaving manner. However, in later discussions we restrict the firing choices for modeling LIS, which is a synchronous design. For the above rule of transition enabling, we assume that each place can hold infinite number of tokens, since only the places in $^\bullet t$ are used to determine the enabling of transitions. For example in Fig. 2.1(a), $T1$ is enabled iff $M(P0) > 0$ regardless of $M(P1)$.

Such an MG models an infinite capacity systems. For actual physical systems, each place p has a capacity $K(p)$, indicating the maximum number of tokens a place p can have. In Fig. 2.1(a), $K(P0) = 2$, $K(P1) = 1$ and $K(P2) = \infty$. With the constraints on capacity, the above firing rule should be changed to take the output place capacity into consideration, (e.g. $T1$ is not enabled if already $M(P1) = 1$) which leads to the *strict firing (transition) rule*. Comparatively the firing rule without considering the place capacities is referred as *weak firing rule*. For the strict firing rule, transitions t is enabled iff: $\forall p \in {}^\bullet t$, $M(p) > 0$ and $\forall p \in t^\bullet$, $M(p) < K(p)$.

One can either directly use strict firing rule for the MG, or still apply the weak firing rule to a transformed MG$'$ derived from MG according to the following steps as mentioned in [61].

1. Add a complementary place p' for each place p, and assign the initial marking of p' as $M_0(p') = K(p) - M_0(p)$.
2. Add arc (p', t) and (t', p') if arcs (t, p) and (p, t') exist. So the sum of $M(p)$ and $M(p')$ is $K(p)$ before and after the firing of t.

In the transformed MG$'$, the tokens on the complimentary place p' can be viewed as the number of available space on p. When $M(p') = 0$, a transition t ($p \in t^\bullet$) is disabled because $M(p) = K(p)$ i.e., p is already full. Also, the situation that place p has infinite capacity can be treated in this framework as well. In this case, $K(p) = \infty$, then it complimentary place in MG' p' has a initial marking $M_0(p') = K(p) - M_0(p) = \infty$. It indicates that $M(p')$ is always non-zero thus the condition related to $M(p')$ is always satisfied. Therefore, whether p' exists or not, does not change the system's behavior and p' can be removed. In the transformed MG$'$, if one place p has no complimentary place, it can be inferred that $K(p) = \infty$ and vice-versa. The derived MG$'$ for weak firing rule of the MG in Fig. 2.1(a) is shown in Fig. 2.1(b). After the transformation, strict firing rule can be enforced equivalently to its weak firing counterpart. In later contexts, we always assume the MG is fired by weak firing rule, since in case of strict firing rule is needed, we can still apply weak firing rule on the modified MG$'$. This basic point must be remembered when we move to the discussion on buffer capacities and back pressure.



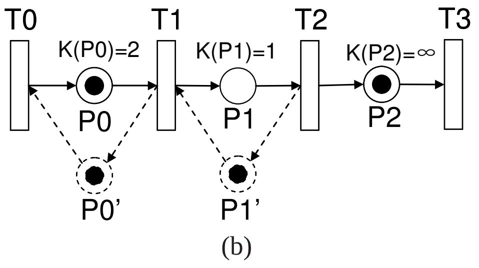


Figure 2.1: (a)An MG;(b)The derived MG$'$ for using weak firing rule

2.1.2 Some properties associated with concurrent system

After a concurrent system is modeled with an MG, a group of behavioral properties can be easily checked against the structure of its MG model. It facilitates the formal analysis of the concurrent system and is one of the major reasons that MG is favored as a modeling notation. These properties normally refer to the *reachability*, *boundedness*, *liveness*, *reversibility*, *coverability*, *persistence* and *safety* as discussed in [61]. Some of these terms are also known in the area of formal verification but they may have different interpretations in the context of MG. Model the properties used throughout this dissertation such as reachability, boundedness and liveness are briefly introduced below. Readers can refer to [61] for elaborate discussions of other properties as well as the various ways to verify them.

1.Reachability
At a particular marking M, some of the transitions are enabled based on the firing rule. Firing one of these transitions t will redistribute the tokens and lead to a different marking M'. In this way, a firing sequence $\sigma = M_0 \overset{t_0}{\to} M_1 \overset{t_1}{\to} \ldots$ gets formed. We say a marking M_n is reachable from M_0 iff there exists a firing sequence σ such that starting from M_0 and arriving at M_n. Usually, $R(M_0)$ is used to represent the set of markings reachable from M_0. It has been shown that the reachability problem is decidable [51, 59] but requires exponential time [63].

2.Boundedness
An MG is said to be k-bounded with an initial marking M_0 iff any marking M reachable from M_0 ($\forall M \in R(M_0)$), the numbers of token on any place p will not exceed k,i.e.,$M(p) \leq k$. Especially, 1-bounded MG is also referred as *safe*. Boundedness guarantees no overflow if the places are used to model buffers and other storages. For an MG, if we use "$M(C)$" to represent the number of tokens residing on a directed cycle C. $M(C)$ is invariant during any arbitrary firing sequences such that $M(C) = M_0(C)$. It can be inferred that if an MG is strongly connected and its initial marking M_0 satisfies that $M_0(p)$ is finite for every places p, then the MG is bounded by M_0.

3.Liveness
Liveness talks about infinite firing. An MG with an initial marking M_0 is said to be live if at the current marking M after firing some transitions from M_0, this process can continue ad infinitum through some further firing sequences. A live MG is deadlock free. If an MG is strongly connected, its liveness can be easily verified by checking $M_0(C)$ for every directed cycle C. If $M_0(C) > 0$, this MG meets the liveness property and is deadlock free (we assume the firings use weak firing rule).

2.1.3 Model LIS using MG

As mentioned above, the firing of an MG has asynchronous characteristic such that only one transition is selected to fire from all the enabled transitions. This is somewhat different from the mechanism of LIS because the latter is a synchronous design such that more than one computation block (either wrapped IP blocks or relay stations) can be executed at the same time. Therefore, we need to synchronize the firings of all enabled transitions to make it suitable for synchronous design. This brings in the notion of ASAP (as soon as possible) scheduling as defined below.

Definition 2 (ASAP scheduling)**.** *For an MG, let $T_{en}(M)$ be the set of transitions that are enabled at marking M. One step of ASAP scheduling is a sequence of transitions which transfers M to a different marking M', such that M' is obtained by firing every transition in $T_{en}(M)$ once. It is denoted as $A(M) = M'$, A maps one marking M to its sequent marking M' by ASAP scheduling.*

This definition indicates that:

1. Each transition can only fire once during one step of an ASAP scheduling. With this restriction, one enabled transition only fires once until all other contemporary enabled transitions in $T_{en}(M)$ are fired. This is due to the fact that in synchronous design, in one clock cycle, the one signal can only be updated once.

2. The system's behavior is only observed every time an ASAP scheduling is executed rather than when just one of the enabled transitions is selected to be fired. One step of an ASAP scheduling can be viewed as one clock cycle in synchronous design. We don't need to consider the firing sequence of transitions in $T_{en}(M)$ during one step ASAP scheduling. Since for an MG, $|p^\bullet| = |^\bullet p| = 1$, there are no transitions reading from or writing to the same place. Firing one transition $t \in T_{en}(M)$ will not disable another transition $t' \in T_{en}(M)$. It can be inferred that M' is fully determined by M based on ASAP scheduling and it is independent of the firing sequence of transitions in $T_{en}(M)$.

Another interesting observation we can made based on ASAP scheduling is the following:

Theorem 1 (Periodic state sequence)**.** *If an MG is strongly connected with initial marking (state) M_0, by ASAP scheduling there exist a marking $M \in R(M_0)$ and a positive integer T_P such that $A^{T_P}(M) = M$.*

Proof. Let an MG m be strongly connected, m is bounded by any firing sequences, including an ASAP scheduling. Therefore, $R(M_0)$ is bounded, i.e., all markings reachable from M_0 are finite. Let $|R(M_0)|$ be the number of markings in $R(M_0)$. At M_0 the next marking M_1 by applying an ASAP scheduling once, is deterministic and $M_1 \in R(M_0)$. If we take this for $|R(M_0)| + 1$ times, there must be two identical markings appearing in the sequences such that $M_i = M_{i+k} = A^k(M)$. This M_i and k are the required M and T_P. □

ASAP scheduling is just a restriction on firing sequences of MG, therefore the properties such as boundedness and liveness of the system are preserved with ASAP scheduling. Theorem 1 tells us that some of the (markings) states are visited periodically. Recall that the markings M determines $T_{en}(M)$, therefore each transition is also fired periodically. In another word, if the MG is strongly connected, each transition can be scheduled with a periodic sequence. This is the fact that a scheduled LIP relies on.

To clarify the following discussions, we now make some simplifications of the notations [61]. Recall that for an MG $|p^\bullet| = |^\bullet p| = 1$. Each place has only one incoming and outgoing arc, we can use place p to represent the link connecting t and t' (where $t \in {}^\bullet p$ and $t' \in p^\bullet$), which consists arcs (t, p) and (p, t'). In later discussions, without specific claim, *we use the term "arc p" for the link between t and t'*.

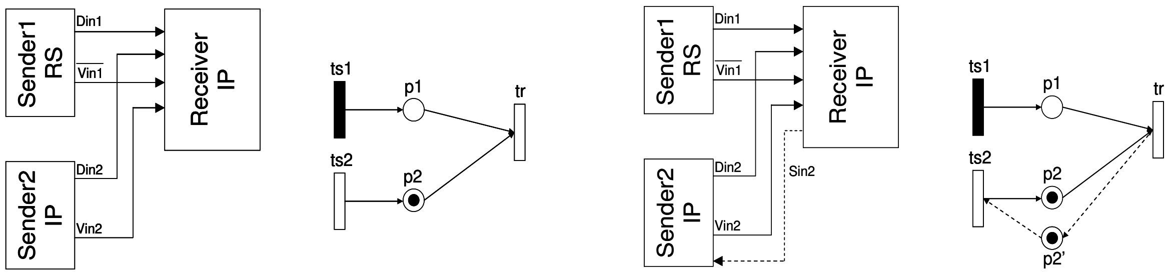


Figure 2.2: (a)A scheduled LIS without back pressure and its MG model;(b)A scheduled LIS with back pressure and its MG model

Let's first model a scheduled LIS without back pressure and then extend it for hand-shake based LIS. The scheduled LIS without back pressure is modeled according to the following rules:

1. The IP blocks and relay stations are modeled by transitions and their storages are modeled by places.
2. Transitions in white and black represent the IP modules and the added RS respectively.
3. Initially, the places related to the output buffer of IP modules are marked by 1 token and others by 0.
4. An MG model is fired with ASAP scheduling.
5. One clock cycle of the scheduled LIS relates to one step ASAP of an scheduling of its MG model. In later discussions, "one clock cycle of an MG$'$' means one step of an ASAP scheduling of the MG.
6. We assume that in LIS, each transition (either an IP or a relay station) has one clock cycle latency (this makes MG become timed marked graph, see the definitions below). If an IP module has multiple clock cycle latencies, we can divide it into several sub-blocks and each will have one clock cycle latency.
7. For a transition, once the transition fires, one token at its outgoing place indicates that the output of its corresponding block is valid, i.e., signal "Valid=1". Otherwise, "Valid=0".

Fig. 2.2(a) shows one part of an LIS without back pressure. The Receiver which is a wrapped IP has two senders, Sender1 is an RS and Sender2 is another wrapped IP. These three components are connected to other components but in this figure only the interconnected signals are shown. Its corresponding MG is shown on the right. Initially, $P2$ has a token meaning that Sender2 produces a valid data (Vin2=1) while $P1$ has no token indicating that the output data of Sender1 is invalid (Vin1=0).

Associating each transition with unit latency makes the MG become a *timed marked graph* (TMG). A TMG is an MG that each transition has a deterministic latency. In the context of modeling LIS, each transition has unit latency. More discussion about TMG can be found in [61]. In the following discussion, we still use the term MG for the model of LIS since TMG is a subclass of MG. The *cycle-mean* of a directed cycle C for an MG is determined by the number of tokens on C over the number of transitions (also the latency of C, denoted as $|T|_C$) on C:

$$\delta(C) = \frac{\sum_{p \in C} M(p)}{|T|_C} \tag{2.2}$$

The throughput of an SCC is determined by $\min_{C \in SCC} \delta(C)$. Previous work such as [19] extended the MG model to Elastic Marked Graph (EMG) to model back pressures.

Definition 3 (Elastic Marked Graph (EMG) [19])**.** *EMG is an MG such that for any place $p \in P$, there exists a complementary place $p' \in P$ satisfying that ${}^\bullet p = p'^\bullet$ and ${}^\bullet p' = p^\bullet$. In addition, at any time $M(p) + M(p') = 2$.*

In [19], a labeling function L is also defined to map all arcs of an EMG as forward or backward ($L : P \to \{F, B\}$) such that $L(p) = F$ iff $L(p') = B$. This complementary arc p' makes p bounded [61] and represents the back pressure. $M(p) + M(p') = 2$ indicates that the forward arc p is bounded by 2. This is coincident with the implementation in [33, 80, 42] where the capacity of each relay station is 2. We define the back pressure arc as follows:

Definition 4 (Back pressure arc)**.** *Back pressure arc of an arc p is an added arc p' to the original MG which satisfies ${}^\bullet p = p'^\bullet$ and ${}^\bullet p' = p^\bullet$, $L(p') = B$ and $M(p') = 2 - M(p)$.*

The arc p in the original MG is called a "forward arc" to distinguish it from the added back pressure arcs ($L(p) = F$). In Fig. 2.2(b), $P2'$ is a back pressure arc added from tr to $ts2$ so $L(P2) = L(P1) = F$, $L(P2') = B$. It corresponds to adding a "stall" (as Sin2 in Fig. 2.2(b)) signal in the original LIS. $M(P2') = 0$ relates to "Sin2=1" which requires that Sender2 be stalled. On the other hand, $M(P2') > 0$ relates to "Sin2=0" which indicates that Sender2 does not have to be stalled.

We need to point out that EMG is just a modification of MG for strict firing rule. In the theory of LIS, both forward latency L_f and backward latency L_b are 1. So in most handshake based protocols, each stage has a buffer size of $L_f + l_b = 2$, which allows storing two data before both the updated "valid" and "stall"signals arrive. If this is directly modeled by MG, each place p should have a capacity of 2 ($K(p) = 2$). With the place capacity, the MG should be fired with strict firing rule. We know that MG can still be fired with weak firing rule if we make some changes to this MG to get MG$'$ as discussed in Section 2.1.1. Not surprisingly, this derived MG$'$ is just the EMG model. As mentioned in Section 2.1.1, if back pressure is not added to one arc p, we assume $K(p) = \infty$.

2.2 Periodic Clock Calculus

It has been proved in the last section that if a system's MG model is strongly connected, transitions which model IP blocks and relay stations with certain periodic patterns. If we relate the firing of one transition t to a binary sequence S_t such that $S_t[k] = 1/0$ denotes at clock cycle k, t is fired/disabled, S_t becomes a periodic sequence $v_t(u_t)$. v_t is the prefix of S_t representing the transient phase and u_t is the part that iterates infinitum. Analyzing the periodic sequence S_t can reveal some properties of the LIS model which can be studied within the framework of periodic clock calculus. Here, we briefly introduce some basics of the periodic clock calculus [29] and we conform to the definition of [29][28][30] and show their applications to analyzing the latency insensitive system schedulings.

Definition 5 (Clock and periodic clock [28])**.** *A clock for an infinite stream is an infinite binary word which is a member of the ω-regular set $(0 + 1)^\omega$. A periodic clock w is an infinitely periodic binary word and can be expressed as: $w = v(u)$, where $(u) = lim_n u^n$ denotes the infinite repetition of u with period $|u|$, and v is a prefix of w.*

Note that if w is the clock of activation sequence (scheduling) for a computation block, then the computation block is activated at the cycle n if the n^{th} letter of w (also denoted as $w[n]$) is 1, else at the n^{th} cycle, the computation block remains idle. For the scheduling purposes, v is often denoted as the transient or initial sequence and u as the periodic sequence. For an arbitrary binary word w, let $|w|$ denote the length of w and $|w|_1$ denotes the number of 1s in w and $|w|_0$ denotes the number of 0s in w. Note that for infinite words, these could be infinite. Let $w[1 \ldots n]$ be the prefix of length n of w. $[w]_p$ denotes the location of p^{th} 1 in w. It can be inferred that:

$$[1.w]_p = [w]_{p-1} + 1, p > 1 \tag{2.3}$$

$$[0.w]_p = [w]_p + 1, p \geq 1 \tag{2.4}$$

Note that if a periodic clock $w = v(u)$ is a clock of a computation block A, then $|u|_1$ refers to the number of firings (executions) of A within one period and $|u|_0$ refers to the latencies of A within one period, therefore the throughput of A can be given by:

$$\delta(A) = \frac{|u|_1}{|u|_1 + |u|_0} = \frac{|u|_1}{|u|} \tag{2.5}$$

Definition 6 (Precedence relation [28])**.** *One can define a precedence relation $\preceq$ between two clocks $w1$ and $w2$, iff $\forall\ p \geq 1,\ [w_1]_p \leq [w_2]_p$.*

This relation is a partial order on clocks e.g. $(1) \preceq (10) \preceq (01) \preceq (001) \preceq (0)$. It can be used to reflect a causal relation between two clocks. For example, if data flow A is dependent on data flow B and w_A, w_B are the clocks of A, B, then $w_B \preceq w_A$.

Definition 7 (Upper/lower bound [28])**.** *The upper bound $w \sqcup w'$ and the lower bound $w \sqcap w'$ of two clocks w and w' can be characterized as sequences with the following properties:*

$$\forall p \geq 1, [w \sqcup w']_p = max([w]_p, [w']_p) \tag{2.6}$$

$$\forall p \geq 1, [w \sqcap w']_p = min([w]_p, [w']_p) \tag{2.7}$$

It could be proved that: $w \preceq w \sqcup w'$, $w' \preceq w \sqcup w'$ and $w \sqcap w' \preceq w$, $w \sqcap w' \preceq w'$. In order to analyze the unification of data flows which have different clocks, [28] discusses synchronizability using *sub-type relation*.

Definition 8 (Synchronizable clocks [28])**.** *Two clocks w and w' are synchronizable, written as $w \bowtie w'$, iff $\exists d,\ d' \in N$ such that $w \preceq 0^d w'$ and $w' \preceq 0^{d'} w$. It means w can be delayed by d' ticks so that the 1s of w' occur before the 1s of w, and reciprocally.*

It is also interesting to note that if for all n, the n^{th} 1 of w is bounded by a finite distance from the n^{th} 1 of w', then a bounded FIFO is enough to "slow down" one data flow according to the other. Further, [30] gives a sufficient and necessary condition of synchronizability for periodic clocks.

Theorem 2. *[30] Two periodic clocks $w = v(u)$ and $w' = v'(u')$ are synchronizable, denoted as $w \bowtie w'$, iff they have the same rate (throughput)*

$$\frac{|u|_1}{|u|} = \frac{|u'|_1}{|u'|} \tag{2.8}$$

Definition 9 (subtyping relation [28])**.** *The subtyping relation, written as $<:$ is the conjunction of precedence and synchronizability: $w_1 <: w_2$ iff $w_1 \preceq w_2$ and w_1 is synchronizable with w_2.*

Therefore, if there is subtyping relation between two clocks, they must have the same throughput. In a computation block, the clocks of the input and output data flows must satisfy this constraint if bounded buffers are used and no data should be lost.

Theorem 3. *In a causal computation block, $si_1, si_2, \ldots, si_N$ are the input signals and $so_1, so_2, \ldots, so_M$ are the output signals. clk_si_k, clk_so_j are the clocks of these signals, $k \in \{1, 2, \ldots, N\}$ and $j \in \{1, 2, \ldots, M\}$. The computation block can work correctly with bounded buffers iff $clk\_so_j = \bigsqcup_k clk\_si_k \Rightarrow clk\_so_j :> clk\_si_k$.*

This is apparent, for example in Fig. 2.3, a and b are the input data flows of the computation block and c is the output data flow. Here we assume the computation blocks are *causal systems*, that the system's outputs only depend on the previous and the current inputs but not the future inputs. clk_a, clk_b and clk_c are clocks of a, b and c, then $clk\_c = clk\_a \sqcup clk\_b$. To make a and b synchronizable, clk_a, clk_b must have the same throughput, otherwise, if throughput of clk_a is larger than the throughput of clk_b, infinite data will aggregate in the input buffer of a and unbounded buffer is required. By $clk\_c = clk\_a \sqcup clk\_b$, it implies that clk_c should have the same throughput as clk_a and clk_b. Combined with the causal relation, it implies that $clk\_c :> clk\_a$, $clk\_c :> clk\_b$. It is used when analyzing the scheduling relations of a join transition (a transition with multiple incoming arcs) for latency insensitive systems. On the other hand, if such type

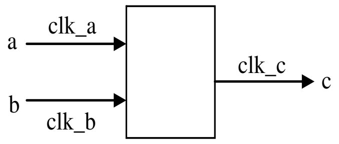


Figure 2.3: A example of subtyping relation

relation are not satisfied between clk_a, clk_b and clk_c, the block cannot compute c correctly with bounded buffers.

2.3 SIGNAL language and Polychronous Framework

2.3.1 SIGNAL language

SIGNAL is a synchronous programming language for real time applications [41], which allows describing a system with concurrency specification and verify the functional correctness without considering the actual implementation. In a SIGNAL process, signals (a signal is defined as a infinite sequence of events) like x,y can update at different frequencies determined by their "clocks", denoted as $\widehat{x}$ and $\widehat{y}$. The clock of a signal indicates the set of instants at which this signal is absent or present. The relations of clocks are implicitly included in SIGNAL operations shown in Table 2.1. For a boolean signal C, $[C]$ refers to the set of instants when boolean variable $C = true$. Clocks can also be explicitly related using clock equations shown in Table 2.2. One can refer to [9] for the detailed explanation of SIGNAL syntax shown in Table 2.1 and Table 2.2.

2.3.2 SDFG and CH

One of the tasks of the SIGNAL compiler is to transform the concurrent specification written in SIGNAL language to a sequential implementation in C language. Therefore a valid schedule of the

SIGNAL operator	SIGNAL expression	Clock relation	Conditional dependency
Function	$Y = f(X_1, X_2, \ldots, X_N)$	$\widehat{Y} = \widehat{X_1} = \ldots = \widehat{X_N}$	$X_i \xrightarrow{\widehat{X_i}} Y$
Downsampling	$Y = X\ when\ C$	$\widehat{Y} = \widehat{X} \wedge [C]$	$X \xrightarrow{\widehat{X} \wedge [C]} Z$
Merge	$Y = X\ default\ Z$	$\widehat{Y} = \widehat{X} \vee \widehat{Z}$	$X \xrightarrow{\widehat{X}} Y \xleftarrow{\widehat{Y}/\widehat{X}} Z$
Delay	$Y = X\ \$\ init\ y_0$	$\widehat{Y} = \widehat{X}$	$X(i-1) \to Y(i), i > 0, Y(0) = y_0$

Table 2.1: Conditional dependencies for kernel operations of SIGNAL language [9]

concurrent specification must be found. SDFG is a formal structure used by the compiler to find scheduling. Each basic operation of SIGNAL has its own *conditional dependency*[62] shown in Table 2.1. SDFG is constructed by composing all of the conditional dependencies [58]. A simple SIGNAL process and its SDFG(bottom left) is shown as in Figure 2.4. A valid schedule forbids *combinational loop* in SDFG. Combinational loop in SDFG refers to the followings.

```
process A= (? integer a,b;  ! integer c,d,e;)
   (| c := a - b
    | d := a + 1 when c > 0
    | e := 2 * b when c <= 0 |)
```

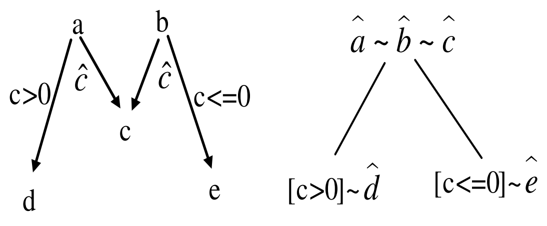


Figure 2.4: SIGNAL description of process A and its SDFG and CH

1. A loop without buffers which is expressed by “$” in a SIGNAL process. For example, in `BX := X $ init 0`, signal `BX` takes the previous value of `X` and which can be treated as a buffer of `X`.

2. A loop with effective conditions. If there is a loop in the path, forexample, $a_1 \xrightarrow{c_1} a_2 \xrightarrow{c_2} \ldots \xrightarrow{c_{n-1}} a_n \xrightarrow{c_n} a_1$, it must satisfy $\bigwedge_{i=1}^{n} c_i$ to be a combinational loop. In other words, all the

Table 2.2: Explicit clock relations using clock equations

Clock relations	Clock equations	Explanation
Synchronization	$Y\hat{}= X$	signal Y is synchronized with signal X
Union	$Y\hat{}=X\hat{}+Z$	values of signal Y are on the instants of signal X or signal Z
Intersection	$Y\hat{}=X\hat{}*Z$	values of signal Y are on the same instants of signal X and signal Z
Deference	$Y\hat{}=X\hat{}-Z$	values of signal Y are on the instants of signal X but not signal Z

conditions along this loop should be true at the same time for the loop to be a true combinational loop. Figure 2.5 shows two processes B, C and their SDFG. Process B has a combinational loop between signal c and d. On the other hand, process C with a slight change has no combinational loop, because there are buffers `zc` and `zd` in the loop.

```
process B= (? integer a,b;
             ! integer c,d;)
    (| c := a + b
     | d := b + c |)
```

```
process C= (? integer a,b;
             ! integer c,d;)
         (| c := a + zd
          |zc := c $ init 0

          | d := b + zc
          |zd := d $ init 0 |)
where
     integer zc, zd;
end
```

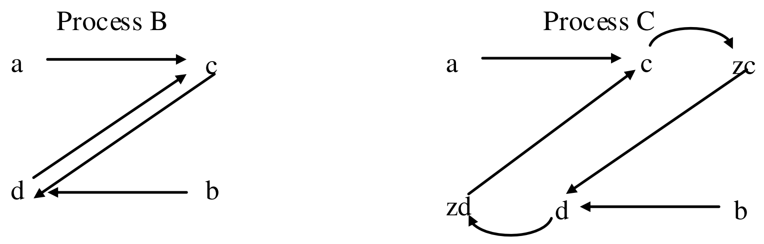


Figure 2.5: Process B with a combinational circle and Process C without combinational circles.

Another important aspect of the compiler is to synthesize the clock relations by a formal structure *clock hierarchy*(CH). In a given CH, any child node clock is a down-sampling of its ancestors. All of the synchronized clocks are grouped as one node in CH. [3] illustrates an algorithm to construct the CH for a given SIGNAL process. The CH of process A is shown in Figure 2.4(on the right) as well. Root node $\widehat{a} \sim \widehat{b} \sim \widehat{c}$ means that signal a, b, c are synchronized and have the fastest clock. $[c > 0] \sim \widehat{d}$ as a child node indicates that clock of d is synchronized with the clock $[c > 0] = true$, and they are downsamples of clock c. Sometimes, there might be contradicting clock relations within one CH, and which is formally defined in [72] as an ill-formed hierarchy.

Definition 2.3.1. *[72] A CH is ill-formed if and only if:*
1. There exists any boolean signal x, that $\widehat{x}$ is a childnode of either $[x]$ or $[\neg x]$, where $[x]$ represents "$when\ x = true$" and $[\neg x]$ represents "$when\ x = false$".
2. There exist signal $b1$, $b2$ and $\widehat{b1}$ is a ancestor of $\widehat{b2}$, where $b1 = c1fc2$ and $\widehat{c1}$, $\widehat{c2}$ are childnodes of $\widehat{b2}$, f $= \{\vee, \wedge, \backslash\}$.

In 1, $\widehat{x}$ is a childnode of its downsamples $[x]$ or $[\neg x]$. In 2, since $b1 = c1fc2$, $b1$ should be a childnode of the common ancestor of $\widehat{c1}$ and $\widehat{c2}$. Here the common ancestor is $b2$, then $b1$ is a childnode of $b2$, and it contradicts that $\widehat{b1}$ is a ancestor of $\widehat{b2}$. Therefore, ill-formed hierarchy results from contradictiing clock inclusion, which is explained by an example given in Chapter 4. SDFG and CH can be used to check communication correctness between interconnected processes.

2.3.3 Endochrony, Isochrony and GALS distribution

Endochrony and Isochrony are properties to ensure correct operation of synchronous processes working in an asynchronous environment. The formal definitions and proofs of endochrony and isochrony can be found in [8] and [7]. In this chapter, we only discuss their applications to GALS and their verification, which is the foundation for our framework of modeling, analyzing GALS design. More rigorous discussion about endochrony and isochrony can be found in [8] and [7].

Endochrony ensures that a process interacts with an asynchronous environment correctly. An endochronous process can uniquely resynchronize a group of data flows to synchronous state transition [8] without external information. An endochronous process has a unique sequential scheduling.

Isochrony enables correct communication between two processes P_1 and P_2. If two processes P_1 and P_2 are isochronous, their communication is independent from the channel delay. In other words, they can behave equivalently as they are interacting through synchronous channels.

GALS refers to a globally asynchronous system made of locally synchronous components, communicating via asynchronous communication media. Endochrony and isochrony provide a sufficient solution for mapping a synchronous program onto a group of distributed processors [6].
Property 1: Suppose we have a system P which consists of a finite collection of processes $P_i(i = 1, 2, 3, \ldots, n)$, P is endo-isochronous if

1. each P_i is endochronous;

2. each pair (P_i, P_j) is isochronous.

If P is endo-isochronous, when each P_i works in an asynchronous environment and communicates with others asynchronously, then P will have the latency equivalent behavior as when all P_is communicate synchronously. Endo-isochrony provides a sufficient solution for GALS design.

Verification of endo-isochrony is employed from [72] and we come up with our framework to model and analyze GALS designs. [72] shows that
Property 2: A process is endochronous iff the process is:

1. without combinational cycle;

2. without ill-formed clock;

3. its CH has a unique root.

Condition 1 can be verified by checking its SDFG and conditions 2 and 3 can be verified during the procedure of construction of the CH. Since it is possible to get the SDFG and CH from a given

SIGNAL process, they can be used to check endochrony for any system modeled by SIGNAL. [72] also pointed out one way of checking isochrony [72]
Property 3: (P, Q) is isochronous, if two P and Q satisfy,

1. $P||Q$, the synchronous composition of P and Q is acyclic;

2. P and Q have no ill-formed clocks;

3. $P||Q$ has no ill-formed clock.

Similarly, condition 1 can be easily checked by analyzing the composition of the SDFG of $P||Q$, condition 2 can be checked by observing CHs of P and Q. We can compose the CH of P and Q to see if there is contradicting clock relations to verify 3. Analyzing the clock relations of two processes is fully described in [43]

2.4 LIS-LOLA, a LIS analyzer

All of our methods related to scheduled LIS design flow are validated on our tool LIS-LOLA, which is a software to analyze the performance of LIS based on LOLA (A Low Level Petri net Analyzer) [67]. LOLA is an open source validation tool for place/transition net reachability analysis. It embeds several state space reduction techniques such as automated detection of symmetries, partial order reduction, sweep-line-method, coverability methods and linear invariants methods. Some of them can also be jointly used. Also, it accepts textual specification as input, which facilitates the integration with other tools. We utilize the LOLA's internal data structures for petri net graphic representation and state space exploring techniques, based on which we have added some features for LIS analysis. The following extensions to LOLA have been done by us:

- LOLA is extended for specification of the ISCAS and the ITC benchmarks. This enables us to access these benchmarks to validate our algorithms and methods in LIS synthesis.

- Simulated latency information can be provided to a constructed marked graph. Additional transitions and related places, arcs, markings are inserted into the constructed graph. This simulates the process of packing shells for IP and adding relay stations in LIS synthesis flow.

- ASAP scheduling is implemented for LOLA. When LOLA is fired, the marking (state) transition conforms to the ASAP scheduling and record the periodic marking sequences as well as firing sequences for each transition. Informations such as throughput, buffer sizes can be reported.

Chapter 3

Related Work

In this chapter, we discuss the past and contemporary important contributions in the field of GALS design styles, formal analysis models for GALS and in particular LIS designs. In this dissertation, we are more interested in modeling and analyzing the GALS design rather than GALS implementation details. So we just give an overview of different existing GALS design styles. We don't look into too much detail about progresses related to physical characteristic, such as ring oscillator's variability and clock network optimization. Rather, we emphasize more on the current formal methods which have been applied to GALS and LIS design. For the LIS design, an introduction about the two fundamental design styles (hand-shaking, scheduling) is given in Chapter 1, here we focus on the techniques which have been proposed for performance optimization.

3.1 Overview of GALS design styles

The term GALS was first used in the work of Chapiro in his doctoral dissertation, "Globally-Asynchronous Locally-Synchronous Systems"(Dept. of Computer Science, Standford Univ.,1984) [24]. In his work, he proposed an interface using pausible clock circuitry. For the last twenty years, many efforts about GALS design have been made both in academia and industry [47]. Broadly speaking, most of these work adopt a similar architecture in which the locally synchronous blocks are packed with some glue logics as interfaces and interconnected through communication fabrics. The central issue of GALS design is to handle the problem resulting from crossing clock domains. A flip-flop or latch sampling data from other clock domains may violate the setup and hold requirements. It leads to an indeterminate output voltage and requires unbounded length of time to become stabilized to a valid value [23], which is regarded as metastability failure. This problem can be handled by using a series of flip-flops connected in a row (Fig. 3.1) and the failure probability drops exponentially with the number of flip-flops [76].

It is the responsibility of the communication fabric to make the correct data transfer between different clock island. Based on the techniques required, the existing GALS designs can be classified

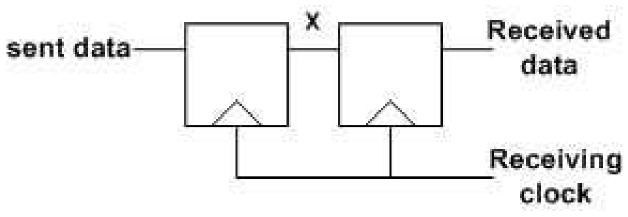


Figure 3.1: A two-flop synchronizer

into three categories: pausible clock, asynchronous and loosely synchronous [76][1].

3.1.1 Pausible clocks

The idea of Pausible clock was proposed by Chaprio in his dissertation in [24]. Pausible clocks are used to decouple clock domains to transfer data safely and to avoid metastability. Each synchronous block can generate its own clock using a ring oscillator and the frequency of which should meet the frequency requirement for this particular block. Control signals are used to stop or restart the ring oscillator which start or stop its own clock. With the ability to stop the clocks, data is transferred between wrappers when both the data transmitter and receiver's clocks are stopped [82].

There are two major advantages for pausible clocking technique. The clock edge of the receiving block can be delayed or stretched by stalling the ring oscillator until the data from the sending block arrives. Thus metastability is completely avoided. On the other hand, shutting down an IP block through the ring oscillator when the IP block is idle (not performing any computation or processing) can reduce dynamic power consumption. However, the disadvantage of pausible clocking is also apparent. Ring oscillator are impractical for industrial use because they are very sensitive to process, voltage and temperature variations. So most designs only choose pausible clocking if power consumption is a critical consideration.

3.1.2 Asynchronous interfaces

An asynchronous interface uses circuits known as synchronizers to transfer signals such as *request* and *acknowledge* hand-shaking signals, from the outside clock domain to the local clock domain. Synchronizers have circuitry similar to the one shown in Fig. 3.1 to reduce the possibility of metastability failure. The hand-shaking signals guarantee correct data transfer across the clock domains. As a result, the received data flow is without duplicated or missing data compared to the transmitted data flow. This approach is reliable but suffers from low throughput.

[68] increased the throughput of an asynchronous interface by pipelining the synchronous operations through a FIFO buffer along with the data. [25] provides a low-latency synchronizing FIFO

[1]Other classification can be found in [52]

buffers by detecting when the FIFO buffer is nearly empty (at the transmitter's side) or nearly full (at the receiver's side). These predicative information are used along with the traditional full and empty signals to control the data flow adaptively.

Another approach for asynchronous interface is realized by asynchronous FIFO between the transmitter and receiver [5]. The FIFO is designed with special treatment which hides the synchronization problem inside the FIFO buffers. It is reported to tolerate large interconnect delays and resilient to metastability. The advantage of this design is that it takes the FIFO out of the synchronous blocks thus avoiding any changes inside the synchronous blocks. However, the FIFO buffers normally have wide interconnect data buses and can cost a lot of area.

3.1.3 Loosely synchronous interfaces

Loosely synchronous interfaces apply to the situation when the frequencies difference or their bounds between the communicating blocks are known before hand. In this situation, the handshaking can be removed if the designers are able to exploit these bounds to ensure that the timing requirements are met. So the resulting system are able to achieve higher performance and have more deterministic latencies than those of the asynchronous interfaces. The most common case is a *mesochronous* relationship, where the transmitter and receiver work on exactly the same frequency but their clocks have an unknown but steady phase difference. It usually happens when the two clocks are derived from a same source but deliveries to the two blocks have different latencies in the clock network. Self-timing scheme at the receiver's input [40] is used so that a self-timed FIFO buffer in the receiver can compensate for the phase difference. The FIFO buffer is properly initialized to *half full*. During operation, the number of data in FIFO remain within ± 1 data item of half full since the sending frequency and the receiving frequencies match. In this case, stopping signals are unnecessary hence the throughput is increased. [22] provides an interface with single stage, clocked FIFO buffer, which tolerates nearly two clock periods of phase uncertainty between the transmitter and the receiver.

3.2 Formal approach for GALS and LIS design

In this section, we discuss several formal approaches which have been applied to model and analyze GALS and LIS. Both GALS and LIS are concurrency systems. They consist of multiple components which may execute simultaneously with interactions. Issues like indeterminacy, deadlock and overflow are potential risks for such systems if they are not designed properly. Formal methods can play an important role to help designer find reliable solutions for coordinating the execution, data exchange, buffer allocation, and execution scheduling to minimize response time and maximize throughput. Models of concurrent system have gained many attractions from researchers and designers in the field of GALS and LIS studies. In this section, we introduce some related works on using formal approaches for GALS and LIS design.

3.2.1 Kahn processing network (KPN)

Kahn process network (KPN) is a distributed model computation (MoC) where each node of the network is a sequential deterministic process and it is communicating with other nodes through unbounded FIFO channels [44]. Regardless the computation varieties and communication delays, the process network is guaranteed to be deterministic. KPN was initially proposed to model distributed systems but it is also found many applications in modeling embedded systems, signal processing system and other high-performance computing systems. Processes exchange data via unbounded FIFO channels. Writing to a channel is non-blocking, while reading from a channel is blocking. A process can't test input availability but can only stall there waiting for a token if that input channel is currently empty.

[70] had proposed a GALS design methodology based on KPN MoC and which is shown in Fig. 3.2. From a the system's description, the behaviors of the system is identified as a collection of concurrent processes communicating asynchronously with a Kahn Process Network (KPN) MoC. Then, the specification can be validated via functional simulation to ensure the correctness of the KPN model. Transformation from the KPN model to the target GALS design is made and the refinements use a components library.

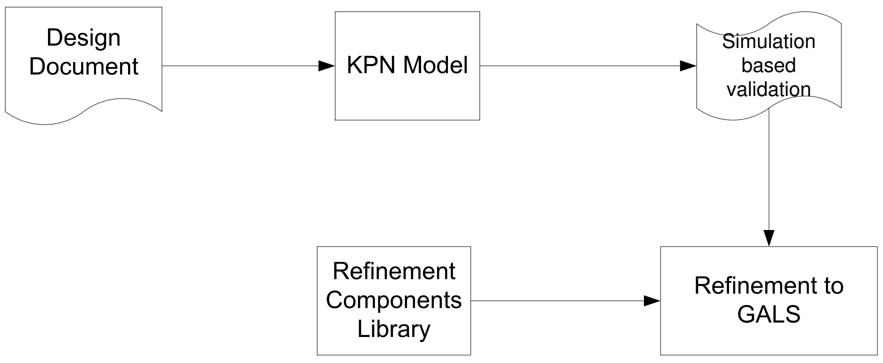


Figure 3.2: A flow of GALS design methodology based on KPN [70]

In this design methodology, [70] created GALS design a correct-by construction design by refining a its specification as a KPN. The refinement transformed each process with a blocking read and non-blocking interface. The communication between the refined processes is facilitated by asynchronous communication with a shared on-chip memory. Meanwhile, [70] proved that their refinement preserve the KPN properties using I/O automata such that each process is deterministic, continuous and monotonic. Also, they showed that the KPN model and the refined GALS model are latency equivalent by firstly showing that each process and its refinement is latency equivalent, and then showing that composition of processes preserves latency equivalence.

Bounded Dataflow Networks BDN is a variation of KPN, where the FIFO sizes are bounded. Nodes can enqueue into a FIFO only when the FIFO is not full and dequeue from a FIFO only when it is not empty [44]. [77] presents a theory for modular refinement of synchronous sequential circuit (SSM) using BDN. They propose the LI-BDN, a special class of BDNs with some good compositional properties and provide a flow for refinement of any SSM into an LI-BDN and the latency insensitive property is preserved under parallel and interactive composition. Also, the LI-BDN can be extended to refine multiple-cycle computation block with correct function.

3.2.2 Process calculi

Process calculi, also referred as process algebras, facilitates system-level description of interactions, communications, and synchronizations between a collection of concurrent processes. It also permits formal reasoning about equivalences between processes (e.g., using bisimulation), thus being widely used in specification and validation of GALS, LIS design.

1. CSP

Communicating Sequential Processes (CSP) is a formal language for describing patterns of interaction in concurrent systems [64]. CSP provides two classes of primitives in its process algebra: Events and Primitive processes. Events represent communications or interactions and which are assumed to be indivisible and instantaneous. They can be atomic names, compound names and input/output events. Primitive processes represent fundamental behaviors of processes.

A process alphabet consists of input (A) and output (B) which are called the input and output channels. An input value v arriving at channel c is denoted by $c?v$ and $c!v$ represents a output v released by c. Basic behaviors of processes can be expressed by CSP operators.

1. **Deterministic choice**: $a \rightarrow P \Box b \rightarrow Q$. It represents the process that waits for event a or b and will transit to process P or Q if it gets a or b.
2. **Parallel composition**: $P \parallel Q$ denotes that processes P and Q synchronize on common actions, dependencies exist between P and Q.
3. **Interleaving**: $P|||Q$ represents that processes P and Q can execute independently of each other and no interaction on any events between them except termination.
4. **Conditional actions**: $P \lhd cond \rhd Q$ represents that P is performed if the condition $cond$ holds. Otherwise Q is executed.
5. **Others**: $skip$ is an event that happens instantaneously, $stop$ makes one process deadlock, $error$ denotes a divergent state.

[45] presents CSP model for LIS. It models the computational blocks and connectors for LIS. The composition of IP blocks is done by the parallel composition. Also, properties such as liveness and deadlock freedom of LI systems are given as conditions for later check. In this model, [45] compares streams of events from the synchronous model of the system against its latency insensitive

implementation by checking bisimulation and the latency equivalent is found to be satisfied. The detailed model descriptions can be found in [45].

2. CCS

The Calculus of Communicating Systems (CCS) was introduced by Robin Milner around 1980 in his book with the same title [60]. It models indivisible communications between exactly two participants, which includes primitives for representing parallel composition, choice between actions and scope restriction. CCS is useful for evaluating the qualitative correctness of properties of a system such as deadlock or livelock [49].

[69] models each component of GALS designs in CCS processes which supports nondeterminism and provides a unique formal verification using bisimulation semantics for equivalent checking between specification and implementation. [46] presents a method to describe both the specification and implementation of delay-insensitive (one kind of GALS interfaces) in CCS. Particularly, the property of delay-insensitivity is also defined as a *MUST-testing* preorder in CCS which captures the refinement relationship between specification and implementation.

3.2.3 Petri-net

In Chapter 2, an introduction of marked graph (MG) is given, which is a special case of Petri-net. The definition of Petri-net is somewhat similar to MG but is more general that one place p can have multiple incoming and outgoing arcs. Two subsets of Petri-net gain the most popularity in modeling GALS and LIS designs, MG and *signal transition graph*(STG). The first one is usually used to model the overall communications network of the system while the latter is good candidate to represent the asynchronous interface. Here we give a brief introduction of STG. On the other hand, since we have already introduced MG in the last chapter, we will focus some variation of MG for modeling GALS and LIS.

1. Signal transition graph (STG) An STG $\Gamma(\Sigma, Z, \lambda)$ is defined based on a Petri-net $\Sigma(P, T, M)$ that Z is a finite set of binary signals, which generates a finite alphabet $Z^{\pm} = Z \times \{+, -\}$ of signal transitions. $\lambda : T \rightarrow Z^{\pm}$ is a labeling function. In STG, each node represents a transition of a signal. For signal x, x^{-} denotes x is changed from “1” to “0” (falling edge) while x^{+} denotes the change from “0” to “1” (rising edge). STG is the most widely used model to specify the asynchronous circuits. One can find many work use STG to represent the systems for synthesis and verification [26, 65, 79, 81].

2. Marked graph and its variations MG and its variations has been successfully used in LIS modeling and analysis [20, 31, 57, 78, 12, 13, 11, 32, 17, 15, 19]. For most of these works, timed marked graph (TMG) is used where each transition has a unit latency which expresses one clock delay of each module. [19] uses elastic marked graph based on TMG is proposed, such that each arc has an complementary arc with the opposite direction to the data flow, which models the backward stall signal. [15], an similar MG variation denoted as back pressure graph (BPG) is proposed. The only difference is that the capacity of an forward arc can be increased. This is used for throughput improvement through buffer resizing (see the discussion in Section 3.3.1). In [32], the dual marked graph (DMG) is provided by extending MG by allowing negative markings and early firing, which enables the early evaluation (see the introduction in Section 3.7).

3.2.4 Other formal methods for verification

A variety of formal verification methodology have been developed and applied for GALS, LIS designs. Most of these methods use model checking technique which verifies the protocol by exploring all of the reachable states.

Rostislav et. al. [36] propose a way of converting signal transition graphs of asynchronous protocols into PSL statements and employ assertion based verification tool to do complete verification. In [50] , a verification framework based on process space is employed, where a new data transition model representing the implicit relationship between clock and data validity events is proposed and a comprehensive implementation models for the asynchronous wrapper and the asynchronous communication scheme is constructed; In [35], Dasgupta et. al. propose deadlock tolerant GALS ring architecture using Petri net specification and used model checking tool for reachability analysis and deadlock check. [66] verifies the asynchronous hardware designs specified in CHP, a VLSI programming language based on process calculi, using exiting model checking tool CADP that are based on exploration of LTSS (labeled transition system). Specially for LIP, [71] propose a framework to validate the families of LIP using Spin model check for latency equivalence. It also models the LIP with SML to validate the functional correctness by a programming based simulation technique.

3.3 Performance Optimization for LIS

Another aspect of LIS research is on performance optimization for LIS design. Many works are there exploring various ways of improving the design's performance, especially the throughput. This section gives a brief discussion of these approaches.

3.3.1 Performance optimization by slack matching

In hand-shake based LIS, the performance maybe degraded because of using stall signals. Fig. 3.3 is an MG model of LIS using hand-shaking from [19]. As discussed in Section 2.1.3, the throughput of the original system (without considering the back pressure arcs) is 3/5, determined by the cycle $a \to b \to c \to d \to e \to a$. However, if hand-shakings are used for every link and each arc has a capacity of 2 [16], the throughput is brought down to 1/4 by the cycle $h \to j \to i \to h$. At h, waiting for token from d slows down the system. A method called *slack matching* is used to make the token come to h with a small latency difference. Slack matching achieves this by increasing the capacity of short branch (the incoming path which has less latency) with the aim of balancing fork-join branches [18].

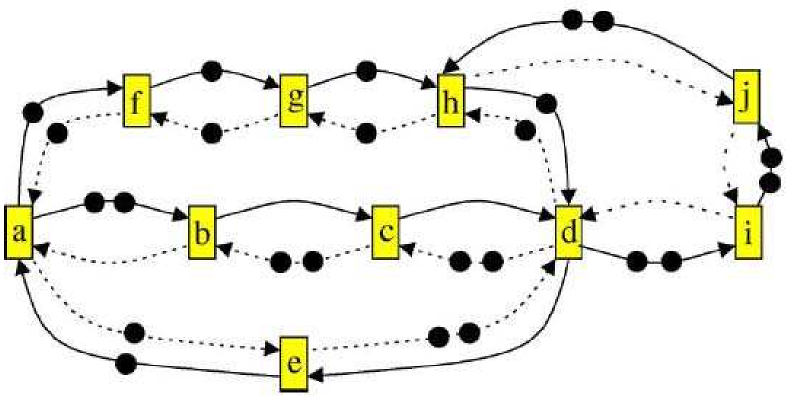


Figure 3.3: One example showing the throughput degradation [19]

There are two technique that can be used to increase the capacity and are referred as *Buffer sizing* and *Recycling* according to [18].

- **Buffer sizing** is used to increase the capacities of buffers. Intuitively, increasing the buffer capacity of one arc p (see the assumption we made to simplify the expression in Section 2.1.3) can increase the initial tokens on its complementary arc p'. If p' is on a critical cycle, adding tokens on this arc can make this cycle no longer critical or increase the throughput. For instance, in Fig. 3.3, if one of the forward arcs in $h \to j \to i \to h$ increases its capacity by 1. Then the throughput will be increased to 2/4. If another arc in this cycle increases its capacity further by 1, the throughput will be raised to 3/4 and resulting system becomes the one shown in Fig. 3.4. Generally, the minimum additional buffer sizes required to achieve a given throughput can be found by solving a mixed integer linear programming formulation (MILP) which is known NP-complete. [56]. [56] is the first work, according to our knowledge, to provide such formulation. It restricts the throughput by restricting every cycle mean (see definition in Section 2.1.3), which again can be expressed by a group of linear inequalities in terms of rational number associated with the transitions. [15] extends this formulation for asynchronous design.

- **Recycling** is to add bubbles in the short branches, the one with short latencies. This technique is based on the idea that breaking down a long combination path can increase the clock period. Therefore, recycling is similar to the idea of buffer insertion in retiming [54]. For Fig. 3.3, a bubble is inserted in arc $j \to h$ by adding additional transitions. The original arc $j \to h$ is broken into two arcs which achieves the same throughput as obtained by buffer sizing shown in Fig. 3.5. [18] pointed out that recycling can't always achieve the same throughput improvement as buffer sizing.

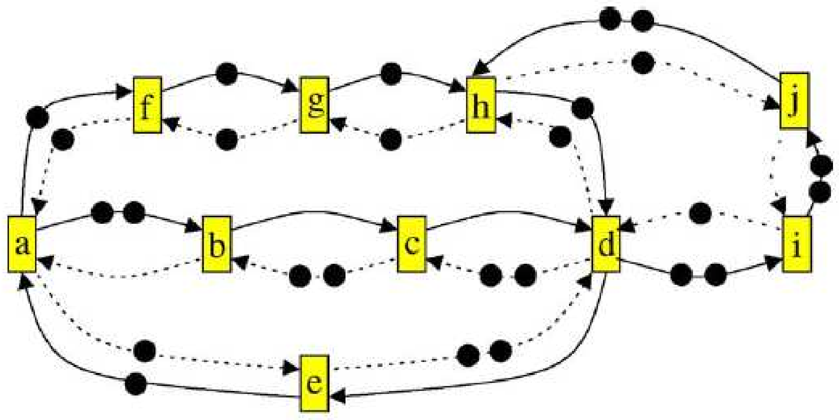


Figure 3.4: Throughput improvement through buffer resizing [18]

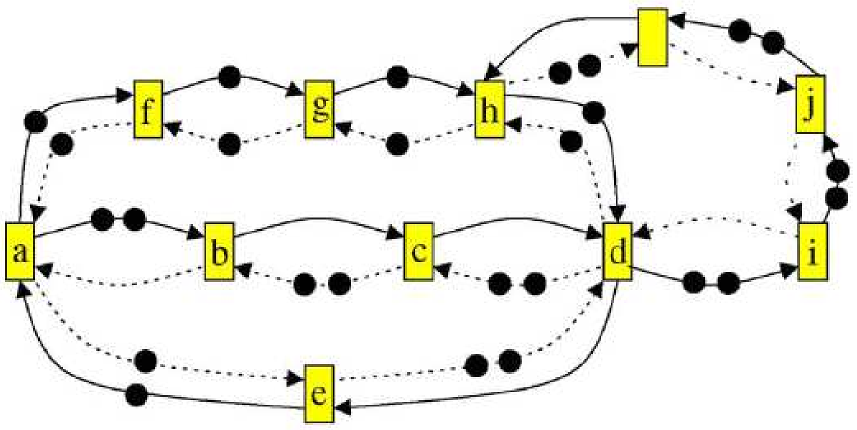


Figure 3.5: Throughput improvement through recycling [18]

3.3.2 Performance optimization by exploiting dependencies

LIS is normally designed without considering the information inside each IP block to keep the integrity of the predesigned IP modules, thus each IP block is treated as a black box. However, in some cases if the internal information such as dependencies within one block is known for the designer, it can be used to reduce the implementation of the LIS and globally optimize the system's performance. This is studied in [55]. Since IP blocks are synchronous systems, they can always be expressed as FSMs. In [55], the functional independence condition (FIC) is obtained before synthesis of the LIS system. FIC indicates whether in state S_i, the state transition to S_j or the generation of output O_k is independent of a particular input I_l. If it is independent, in S_i the FSM does not need to be stalled waiting for the input I_l. Both the wrapper design can be simplifed and the throughput can be increased.

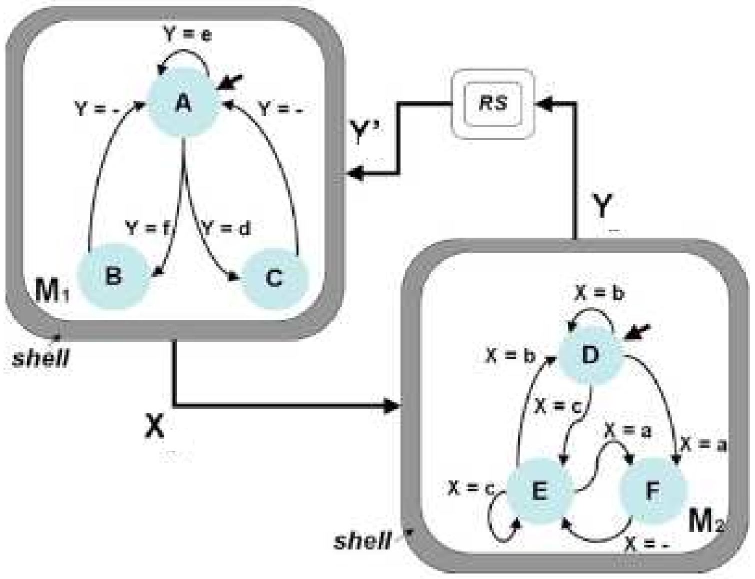


Figure 3.6: An motivating example

Consider a motivating example from [55] shown in Fig. 3.6. A relay station is inserted on the data path of Y. Two synchronous IP blocks communicate with each other through signal X and Y [55], whose FSMs are $M1$ and $M2$. Each FSM has an output variable and is just the input variable of the other FSM. $M1$ has one output X which is also the input of $M2$. $M1$ has one input Y which is the output of $M2$. The transition arc is marked by the condition which activates the transition. Both $M1$ and $M2$ have three states $M1 : \{A, B, C\}$, and $M2 : \{D, E, F\}$. $M1$ and $M2$ are single output Moore FSMs. We assume that in each state S, the output variable is equal to the corresponding lowercase letter s, e.g. $M1$ outputs $X = a$ in state A, $X = b$ in state B, $X = c$ in state C, so is Y. $M1$ has an initial state of A and $M2$ has an initial state of D. In state C of $M1$, transition to A is independent of its input Y. Also in state B, transition to A is independent of Y. For $M2$, transition from F to E is independent of input X. FIC detectors at the shells are developed in order to obtain this information, based on which the system's throughput can be increased from 0.66 to 0.72 as claimed in [55] by removing unnecessary synchronization on these non-required inputs.

3.3.3 Performance optimization by early evaluation

Traditionally, the convergent block, the block has multiple inputs can be enabled only when all of its input data are available. This is often referred as late evaluation and is too strict especially when the reconvergent block plays a role as a multiplexer. For example, consider a block that takes three

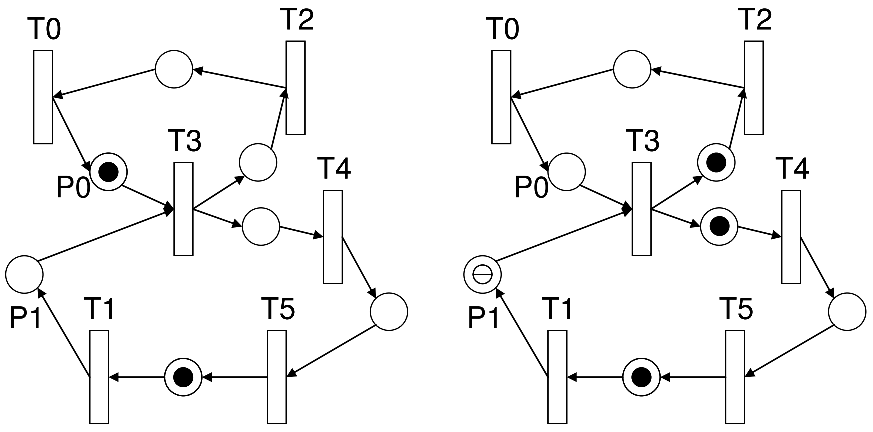


Figure 3.7: An LIS with early evaluation

inputs $in1$, $in2$ and S, where the first two are input data and S is a choice signal.

$$out = \begin{cases} in1 & \text{if } S = 1 \\ in2 & \text{if } S = 0 \end{cases} \tag{3.1}$$

Such a block can be controlled by an early evaluation of the expression. For instance, output can be generated right away as $out = in1$ if $in1$, S are available and $S = 1$ is evaluated. It does not have to wait for the availability of $in2$. At the same time, data $in2$ should be dropped if the output has already been generated by early evaluation which prevents this $in2$ from producing new output.

According to [32], the key problem of early evaluation is to prevent the spurious enabling of a block when the non-required inputs (such as $in2$ in the last example) arrive later after the computation completed. *Antitokens* also termed as negative tokens which were introduced to solve this problem. Comparatively, positive token refers to any token in the original graph. When an early evaluation happens, a negative token is generated at the "don't care" inputs. When they meet the positive tokens on these "don't care" links later, they annihilate, thus the late arriving non-required tokens can not enable the computation block anymore. There are two types of antitokens, *passive* and *active* depending on their "mobility". A passive antitoken by its name, can't move, but only waits for the positive token and is simpler to implement. On the contrary, the active antitoken can travel backward to meet the positive token. It has advantages in terms of power and performance. [18].

With early evaluation, the "AND" firing rule for the reconvergent block should be changed to the "OR" firing rule. The conditions which can enable a block are denoted as guards. There might be multiple guards for a block. The guards for the earlier example are $in1, S$ and $in2, S$. The availability of either guard is enough to generate out. Fig. 3.7 shows an example of early evaluation. Let the guards of block $T0$ be $P0$ and $P1$. With early evaluation, $T0$ is fired when $P0$ has a token and generates a antitoken on $P1$ to destroy its late incoming token.

With early evaluation, the system's throughput can be improved. Several approaches have been proposed for early evaluations to implement generation of antitokens in the flow opposite to data transfer [21, 32].

Chapter 4

Modeling and Analyzing LIP implementation Using Polychronous Framework

As Globally Asynchronous and Locally Synchronous (GALS) based System-on-chip (SoC) are gaining importance, a special case of GALS when the global clocking is preserved, but the interconnect delays of multiple clock cycles are to be tolerated has also been proposed, and used. In this case, the protocols are complex, and many optimized implementations of such protocols need verification that indeed they work correctly with respect to the specification of the system. Usually the specification of the system is fully globally clocked with negligible interconnect delays, so that the specification can be first implemented as synchronous design with traditional tools. GALS or LIP refinements are then applied to tolerate the multi-cycle interconnect delays, or fully asynchronous interconnect communication, as the case may be. Verifying that such refinements are correctness preserving, researchers have used model checking in the past. In this chapter, we present static analysis based framework for such verification. Our framework makes use of the Polychrony framework and associated semantic analysis techniques, in the form of endo-isochrony (see the introduction in Section 2.3.2). We show that the SIGNAL language and its related formal structures can be used to model LIPs and to verify their implementation using static analysis avoiding state space generation based methods such as model checking. With the SIGNAL syntax, the elementary hardware such as combinational logic, sequential logic and FSM can be correctly specified. It enables the modeling of more complicated LIP. On the other hand, two formal structures synchronous dependence flow graph (SDFG) and clock hierarchy (CH) constructed from SIGNAL description, can be used to analyze endo-isochrony and therefore verify the correct communication. These facts provide us an appropriate framework to model and verify the communication of LIP and which is demonstrated and validated through a case study. Analyzing a number of LIP protocols to show whether they preserve the correctness with respect to their fully synchronous specifications using our framework, we believe, designers can verify LIP implementations with clever optimizations using our framework much more readily than when using model checking.

4.1 Modeling Register Transfer Level (RTL) elementary hardware using the SIGNAL language

It is well known that SIGNAL is developed for modeling embedded software. However, most LIPs' implementation are described at lower level such as gate or RTL level. In this section, we describe how the SIGNAL language can effectively model hardware described at gate or RTL level as well. Based on this fact, it is reasonable for us to model more complicated LIP implementation. Generally speaking, the basic components of hardware design include combinational logic, sequential logic such as registers and latches, and sometimes Finite State Machine (FSM) to describe control logics. In the following context, we use SIGNAL to describe combinational logic, latches and registers and FSM respectively.

4.1.1 Modeling combinational logic

Combinational logic is a type of logical circuit whose output is a pure function of the present input only, which can be expressed as a triple $\{I, O, F\}$. I is the set of inputs, O is the set of outputs, F is the set of functions that can be formed by $\{AND, OR, NOT\}$ or their variants. For $\forall o \in O$, $\exists f \in F$ and $\exists i_1, i_2, \ldots, i_m \in I$, $o = f(i_1, i_2, \ldots, i_m)$. Modeling combinational logic with SIGNAL is direct. Each variable in $\{I, O\}$ can be modeled as a boolean or integer signal. Each function in F can be modeled using boolean or arithmetic functions.

4.1.2 Modeling latches and registers

Latches and registers can store their previous value until the trigger conditions are satisfied. Here we use latch to denote level-triggered storage and register for edge-triggered storage. In Figure 4.1, a register triggered at positive edge and its SIGNAL description is shown. Internal signal `ZDout` is defined as the previous value of `Dout` to model the storage. "`when (clk and not zclk)`" expresses the positive edge-triggered condition, where `zclk` is the previous value of `clk`. One can use "`when clk`" ("`when not clk`") to express a high (low) level-triggered condition. "`Dout := Din when (clk and not zclk) default ZDout`" assigns `Dout` to `Din` when the trigger condition is true, otherwise it will take its previous value `ZDout`. `Din^= Dout^= clk` synchronizes the three signals. (i.e, as clk changes value, the value of Din is sampled, and the value of Dout is computed).

4.1.3 Modeling FSM

The finite state machine (FSM) is generally used for control logic, which can be implemented by combinational logic and registers. However, it will be much simpler to express FSM directly from

```
process register= (? integer Din; boolean clk;
                   ! integer Dout;)
   (| Dout := Din when (clk and not zclk)
      default ZDout
    | ZDout := Dout $ init 0
    | zclk  := clk $ init 0
    | Din ^= clk ^= Dout |)
where
    integer ZDout; boolean zclk;
end
```

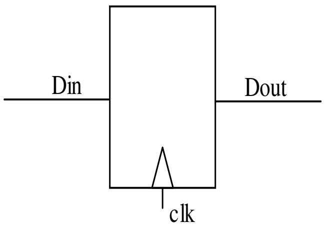


Figure 4.1: A register and its SIGNAL description

its state transition diagram. Figure 4.2 shows an FSM with three states IDLE, REQ and WAIT. In its SIGNAL description, `ack` and `v` are inputs and `req` is the output, `NS` represents next state of FSM and `S` denotes the current state.

```
% State transition of a FSM %
% S:current state, NS:next state %
% S = 0:IDLE, S =1:REQ, S = 2:WAIT %
process FSM = ( ? boolean ack, v;
                ! boolean req; )
  (| NS := 1 when S = 0 when v default
          2 when S = 1 when ack default
          0 when S = 2 when not ack
          default S
   | S := NS $ init 0
   | req := 1 when S = 1 default 0
where
    integer S, NS;
end
```

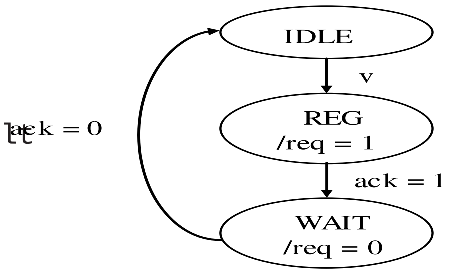


Figure 4.2: A finite state machine and its SIGNAL description

4.2 Modeling LIP using SIGNAL

Now we are going to resend our methodology to model and verify the implementation of an LIP. The methodology has the following steps:

1. Model each component P_i using the SIGNAL language;

2. Construct SDFG and CH of each P_i;

3. Check endo-isochrony of the design by the sets of SDFG and CH;

4. If endo-isochrony is satisfied, the design has correct communication.

We demonstrate this methodology by modeling a particular latency insensitive design in Section 4.2 and analyzing its endo-isochrony in Section 4.3. In the end, the analysis results of other latency-insensitive designs are provided as well.

In this section, we choose to model pSELF proposed by You, et. al. pSELF is a handshaking protocol used for a communication fabric and interface between Intellectual Property (IP) cores, and it traditionally applies to clocked systems. pSELF supports communication with both asynchronous and clocked logics [80]. pEB is one of pSELF protocols as shown in Figure 4.3 from [80]. One pEB consists of two phase elastic half buffers(pEHB) implemented by latches, which

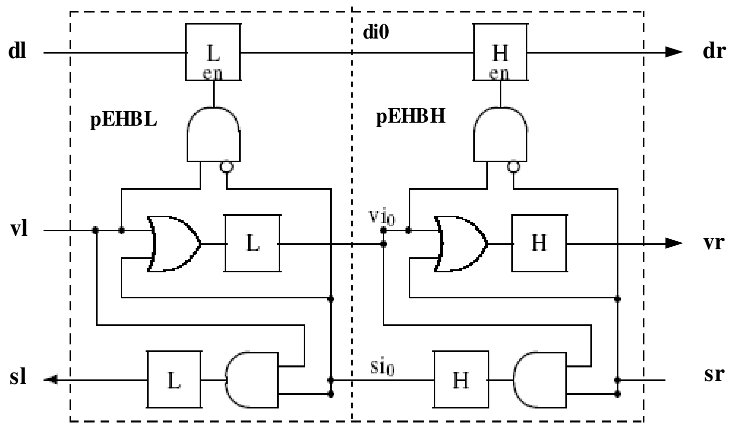


Figure 4.3: Phase synchronous elastic buffer [80]

are enabled on different clock phases. In this figure, the clock signal is not explicitly shown for simplicity. The pEHB with a L-latch is named as pEHBL and the pEHB with a H-latches is named as pEHBH. vl is the valid signal from previous stage to indicate an effective value of input dl, sr is the stall from the next stage to indicate filled buffers and to stop the current pEB generating new outputs. Both vl and sr can propagate through the current pEB. This pEB is modeled as a SIGNAL process pEB and shown in Figure 4.4. Process pEB calls two subprocesses pEHBL, pEHBH with interconnected signals `di0`, `vi0` and `si0`. Here, we only give the code of pEHBL. pEHBH is same as pEHBL, except for latches. In subprocess pEHBL, it firstly synchronizes input signals `Vl, Sr, Clk` and `Dl` by clock equation `^=` since it's a clocked design. Then it generates the outputs of the control signals `Vr` and `Sl` as is quite obvious from the circuit diagram. (`Zvr, Vr`) and (`Zsl, Sl`) model behaviors of the latches in the control path. After that, the enable signal of the data path is generated by the logic. At last, the data path is modeled by `DL` which represents the characteristic of the latch.

4.3 Analysis of Endochrony and Isochrony of LIP using SDFG and CH

In this section, we show the flow of analyzing endochrony and isochrony by a case study of pEB modeled in Section 4.2. In addition, we provide analysis results of other LIP protocols.

4.3.1 Analysis of Endochrony for pEB

Using the SIGNAL compiler description we can extract the SDFG (Figure 4.5) and the CH(Figure 4.6) for process pEB. We need to analyze SDFG to check out combinational loops and CH for ill-formed clocks and single root respectively. To ease the expression, vll and srl are used to represent the internal signals of pEHBL and enl is the enable signal of pEHBL's latch and vlh, srh and enh are named for pEHBH in the same way. Notice that in this SDFG, loops between DL and ZDL, DH and ZDH, sl and zsl, vr and zvr, si0 and zsi0, vi0 and zvi0 are not combinational loops. Another loop vll$\rightarrow$vi0$\rightarrow$srh$\rightarrow$si0$\rightarrow$vll, is also not an effective combinational loop, because the path between vll to vi0 and the path between srl to si0 can not be effective at the same instant. Above all, there is no effective combinational loop in this SDFG. Next, let's move to its CH. As a clocked design, all of the signals in pEB are synchronized with signal Clk and therefore its CH has a special form with only one node. This node is also the root node. It's easy to see that this CH has no ill-formed clock and has a unique root. According to Property 2 in Section 2.3, process pEB is endochronous.

4.3.2 Analysis of composition for two pEBs

Let's move to two pEBs, denoted as pEB1 and pEB2. We can construct their CHs and SDFGs in the same manner and which are named as CH1 and CH2, SDFG1 and SDFG2 respectively. Since both CH1 and CH2 have only one node, the composition of the two will not give any ill-formed clocks, see Figure 4.7. This case is too special and we would like to give readers another counter example to see how ill-formed clock could be generated when composing two different processes. See processes D and E and the composition of their CH's are given in Figure 4.8. s1 and s2 are communication signals shared by both processes. In CH(D) of process D, s2 is a downsampling of s1 and is present when $s1 > 0$. On the other hand, in CH(E) of process E, s1 and s2 are synchronized such that s2 is present when s1 is present. Therefore, composition CH(D) and CH(E) leads to the contradiction that s1 is present when $s1 > 0$ which gives a ill-formed clock.

Next, we look at the composition of SDFG1 and SDFG2 in Figure 4.9. We only need to observe the interface, because we already know from the previous discussion that both SDFG1 and SDFG2 have no combinational loops. Similarly, we can observe that loop vlh→vr→vl→srl→sl→ sr→vlh is not effective and therefore the composition SDFG is acyclic. Above all, we can conclude that the composition of pEB1 and pEB2 are isochronous and therefore a system composed by several pEBs is endo-isochronous and have correct function as if they are working in synchronous environment.

4.3.3 Other analysis results

Apart from the example given in Section 5.1 and 5.2, we have also modeled and analyzed several other LIP protocols listed below. One can refer to the Appendix for the detailed SIGNAL codes. The analysis results are given in Table 4.10

1. Relay station(RS),wrapper(W) from Carloni et. al. [16];

2. Synchronous interlocked pipelines(SIP) from Jacobson et. al. [42];

3. SELF, elastic half buffer master/slave(EHBM, EHBS), fork, join, eager fork from Cortadella et. al. [33];

4. pSELF(pEHBL, pEHBH), interleaving to synchronous protocol(int2syn), synchronous to interleaving protocols(syn2int) from [80].

The results are as expected. All of these LIP listed above are clocked designs and the communication between them are synchronized, therefore isochrony is easily satisfied with the same reason as mentioned in Section 4.3. On the other hand, RS, W, SIP, SELF are based on Elastic Buffer(EB) with handshaking signals *valid* and *stall*, and their structures have much in common. At the same time, EB and pEB are similar. Therefore, it's not surprising to get the same results as Section 4.3.

```
process pEB= (? boolean vl, sr, clk; integer dl;
              ! boolean vr, sl; integer dr;)
   (| (vi0, sl, di0) := pEHBL(vl, si0, clk, dl)
    | (vr, si0, dr)  := pEHBH(vi0, sr, clk, di0) |)
where
    boolean vi0, si0;
    integer di0;
    %subprocesses EHBM, EHBL%
    process pEHBL = (..., ...);
    process pEHBH = (..., ...);
end;

%subprocess EHBL%
process pEHBL = ( ? boolean Vl, Sr, Clk; integer Dl;
                  ! boolean Vr, Sl; integer Dr;)
    %synchronize input signals’ clocks%
   (| Vl ˆ= Sr ˆ= Clk ˆ= Dl
    %Compute Vr and Sl%
    | Vll := Vl or Sr
    | Vr  := Vll when (not Clk) default Zvr
    | Zvr := Vr $ init false
    | Vr ˆ= Vl
    | Srl := Vl and Sr
    | Sl  := Srl when (not Clk) default Zsl
    | Zsl := Sl $ init false
    | Sl ˆ= Sr
    %Date path control%
    | Enl  := Vl and (not Sr)
    %Data path%
    | DL  := Dl when Enl when (not Clk) default DL $ init 0
    | Dr  := DL
    | DL ˆ= Clk |)
where
    boolean Vll, Zvr, Zsl, Zrl, Enl;
    integer DL;
end
```

Figure 4.4: SIGNAL description of pEB

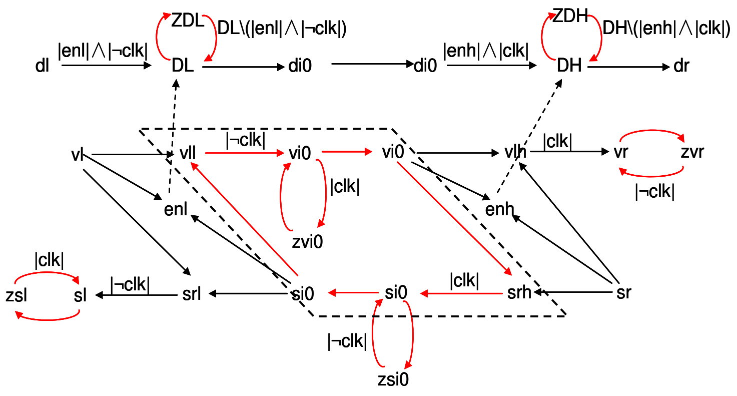


Figure 4.5: SDFG of one pEB

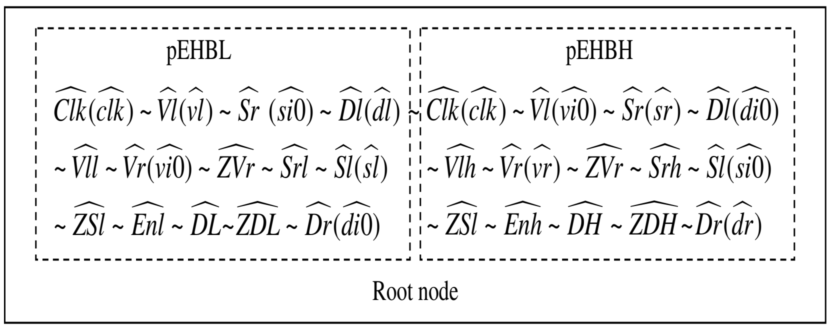

pEHBL
Clk(clk) ~ Vl(vl) ~ Sr (si0) ~ Dl(dl) ~
~ Vll ~ Vr(vi0) ~ ZVr ~ Srl ~ Sl(sl)
~ ZSl ~ Enl ~ DL~ZDL ~ Dr(di0)
pEHBH
Clk(clk) ~ Vl(vi0) ~ Sr(sr) ~ Dl(di0)
~ Vlh ~ Vr(vr) ~ ZVr ~ Srh ~ Sl(si0)
~ ZSl ~ Enh ~ DH ~ ZDH ~Dr(dr)
Root node

Figure 4.6: CH of one pEB

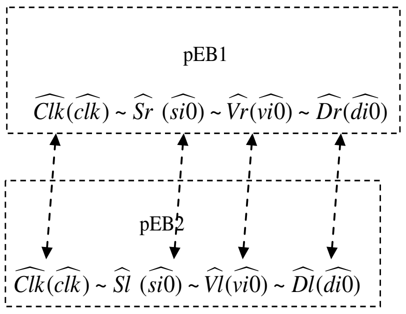

pEB1
Clk(clk) ~ Sr (si0) ~ Vr(vi0) ~ Dr(di0)
pEB2
Clk(clk) ~ Sl (si0) ~ Vl(vi0) ~ Dl(di0)

Figure 4.7: The composition of CH1 and CH2

```
process D= (? integer a,b;              process E= (? integer s1,s2;)
             ! integer s1,s2;)                        ! integer c;)
   (| s1 := a + b                          (| c := s2 + s1 |)
    | s2 := a + 1 when s1 > 0 |)
```

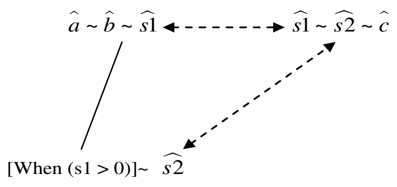

$\hat{a} \sim \hat{b} \sim \widehat{s1}$
$\widehat{s1} \sim \widehat{s2} \sim \hat{c}$
[When (s1 > 0)]~ $\widehat{s2}$

Figure 4.8: Composition of process D and E has ill-formed clocks

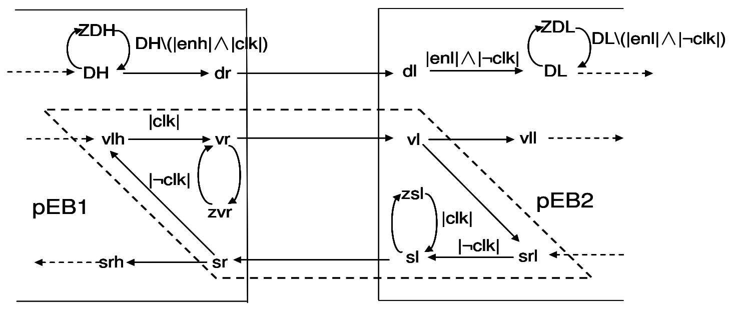


Figure 4.9: composition of SDFG1 and SDFG2 of pEB1 and pEB2

Protocols	Endochronous?
RS	yes
Wrapper	yes
SIP	yes
EHBM	yes
EHBS	yes
fork	yes
join	yes
eagerfork	yes
pEHBL	yes
pEHBH	yes

Compositions	Isochronous?
RS——RS	yes
RS——Wrapper	yes
Wrapper——Wrapper	yes
SIP——SIP	yes
SELF(EHBM——EHBS)	yes
pSELF(pEHBL——pEHBH)	yes
int2syn(pEHBL——EHBM)	yes

Figure 4.10: Analysis results of other LIPs

Chapter 5

Analysis of Latency Insensitive System Using Periodic Clock Calculus

Originally the Latency Insensitive Protocols (LIP) were invented to make a system elastic to the interconnect latencies using handshaking signals such as ‘valid’ and ‘stall’. Such protocols require extra signals leading to area overhead and may affect throughput of the system. To optimize away some of these overheads, scheduled LIPs were proposed which replaced the complex handshake control blocks by a scheduling scheme. One can view a scheduled LIP based design as a system where within each strongly connected component of the system, the modules and the relay stations are scheduled by activation signals. These activation signals can be thought of as infinite sequence of ‘1’s and ‘0’s. If such sequences are periodic, one can view them as periodic clocks. Given the advances in periodic clock calculus in the synchronous programming context, in this chapter, we analyze the LIP scheduling problem within the framework of periodic clock calculus. Such analysis provides straight forward algorithms to compute the throughput of scheduled LIP based systems. Within this framework, we also propose a method to synthesize fractional synchronizers. Fractional synchronizers are used to equalize cycles with different throughputs. Our method can determine the numbers and the scheduling sequences of such fractional synchronizers using the periodic clock calculus. In addition, we provide a static estimation of the required fractional synchronizers based only on the system’s structure which is fast and accurate. The notations of periodic clock calculus are introduced in Section 2.2.

5.1 Scheduling analysis using periodic clock calculus

Let’s consider the system shown in Fig. 5.1. $T0$ and $T1$ are two computation blocks connected by three paths $T0 \rightarrow T1$, $T1 \rightarrow T0$ and $T0 \rightarrow T0$. The communication latencies of these paths are 1, 2, 2 respectively. Therefore, $T2$ and $T3$ in black are relay stations inserted to make each link with unit latency. Initial tokens are marked for the immediate outputs of the computation blocks which are $P0$, $P1$ and $P2$. Suppose the scheduling sequences for $T0$, $T1$, $T2$ and $T3$ are $S0$, $S1$, $S2$, $S3$. Let us express these sequences by periodic clocks defined in Section 2.2. With ASAP rule, whether a transition can fire depends only on the tokens of its incoming places, this is the case when stalling signals are not used that a transition can't be stalled by its outgoing transitions. The tokens on these places are either initially given or later produced as a result of firings of their incoming transitions. For example, the p^{th} firing of $T0$ must be in the instant when both the p^{th} tokens, including the initial tokens, of $P4$ and $P3$ are produced. In other words, we have the relation $[S0]_p = max([M_0(P4).S2]_p, [M_0(P3).S3]_p)$ and which is equivalent to $S0 = M_0(P4).S2 \sqcup M_0(P3).S3$. Similarly, $S0$, $S1$, $S2$, $S3$ can be related by Equation (5.1):

$$\begin{cases} S0 = M_0(P4).S2 \sqcup M_0(P3).S3 = 0.S2 \sqcup 0.S3 \\ S1 = M_0(P1).S0 = 1.S0 \\ S2 = M_0(P0).S0 = 1.S0 \\ S3 = M_0(P2).S1 = 1.S1 \end{cases} \tag{5.1}$$

This system of equations is simple and we can infer that: $S0 = 0.S2 \sqcup 0.S3 = 01.S0 \sqcup 01.S0 =$

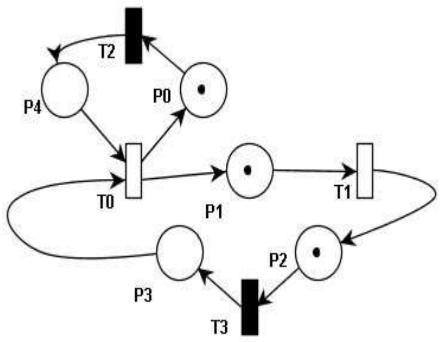


Figure 5.1: An example of strongly connected system

$01.S0 \Rightarrow S0 = (01) = 0(10)$ And then $S1 = 1(01), S2 = 1(01), S3 = 1(10)$. However, generally equation systems such as Equation (5.1) are hard to solve. Instead of solving the equations directly, we use the subtyping relation defined in [28] to infer the constraints on the scheduling sequences. From Section 2.2, Equation (5.1) can lead to the subtyping relations:

$$\begin{cases} 0.S2 <: S0 \quad (a) \\ 0.S3 <: S0 \quad (b) \\ 1.S0 <: S1 \quad (c) \\ 1.S0 <: S2 \quad (d) \\ 1.S1 <: S3 \quad (e) \end{cases} \tag{5.2}$$

According to the definition of subtyping relations, $S0$, $S1$, $S2$, $S3$ should have the same throughput, therefore:

$$\delta(S0) = \delta(S1) = \delta(S2) = \delta(S3) \tag{5.3}$$

By $(5.2a)$, $(5.2d)$: $S0 :> 0.S2 :> 01.S0$ which implies:

$$[S0]_p \geq [S0]_{p-1} + 2 \tag{5.4}$$

This inequality implies that in every two instants, $S0$ can fire no more than once, therefore the rate of $S0$ should be no more than $1/2$. Similarly, $(5.2b)$, $(5.2e)$, $(5.2c)$ implies:

$$S0 :> 011.S0 \Rightarrow [S0]_p \geq [S0]_{p-2} + 3 \tag{5.5}$$

then the rate of $S0$ should be no more than $2/3$. By (5.4) and (5.5), the rate of $S0$ is no more than $1/2$ which is the upper bound for rate of $S0$. Given the fact that the rate of $S0$ is the same as the throughput of $S0$, the throughput of $S0$, $\delta(S0) \leq 1/2$. By (5.3):

$$\delta(S0) = \delta(S1) = \delta(S2) = \delta(S3) \leq 1/2 \tag{5.6}$$

It is interesting to observe that the inequalities $(5.2b)$, $(5.2e)$, $(5.2c)$ and $(5.2a)$, $(5.2d)$ actually relate to the two elementary cycles containing transition $T0$, which are $T0 \rightarrow T1 \rightarrow T3 \rightarrow T0$ and $T0 \rightarrow T2 \rightarrow T0$. Inequality 5.4 gives the upper bound of the throughput for $T0 \rightarrow T2 \rightarrow T0$ and Inequality (5.5) gives the upper bound of the throughput for $T0 \rightarrow T1 \rightarrow T3 \rightarrow T0$. Therefore our analysis is compatible with the problem of finding the minimum throughput of all elementary cycles [53]. It is also observed that the subtyping relation $(5.2a)$, $(5.2b)$, $(5.2c)$, $(5.2d)$, $(5.2e)$ give the scheduling constraints of the marked graph. For example by $(5.2d)$,

$$1.S0 <: S2 \Rightarrow [S2]_p \geq [S0]_{p-1} + 1 \tag{5.7}$$

Equation (5.7) implies that the p^{th} firing of $S2$ should be no earlier than the $(p-1)^{th}$ firing of $S0$ plus one instant, which gives the same relation presented in [14]. Therefore, the schedule of this marked graph can also be obtained by Bellman-Ford's algorithm to find the longest paths to and from a chosen transition. Therefore, this example illustrates that the information used to express the schedule constraints can also be fully expressed by the periodic clock calculus. We extend this discussion later.

Now, we formalize the scheduling problem using the periodic clock calculus for the general situation. Let the marked graph (P, T, W, M_0) be a representation for a scheduled latency insensitive system, we provide some definitions for this formalization.

Definition 10 (Predecessor)**.** *We define transition t_j as a predecessor of transition t_i iff there exists a place p that the token at p is produced by t_j and consumed by t_i. $Pred(t_i)$ is the set of predecessors of t_i.*

Let p_j be the place located between transitions t_j and t_i where $t_j \in Pred(t_i)$ and S_i be the scheduling sequence of transition t_i, N the number of transitions. Then we have:

$$S_i = \bigsqcup\nolimits_{t_j \in Pred(t_i)} M_0(p_j).S_j \ \ (i = 1, 2, \ldots, N) \tag{5.8}$$

We define the *initial scheduling matrix* to represent Equation (5.8).

Definition 11 (Initial scheduling matrix)**.** *We define the initial scheduling matrix is a matrix* $A = \{a_{ij}\}$ *such that:*

$$a_{ij} = \begin{cases} M_0(p_j) & t_j \in Pred(t_i) \\ X & otherwise \end{cases} \tag{5.9}$$

$a_{ij} = X$ *denotes that transition* t_j *is not a predecessor of transition* t_i*.*

Then Equation (5.8) can be rewritten as:

$$S = AS, S = (S_1, S_2, \ldots, S_N)^T \tag{5.10}$$

If we consider the multiply operation as “catenation” and the add operation as the clock upper bound operation “$\sqcup$”, we also regulate $X.S_j = X$ and $X \sqcup a_{ij}.S_j = a_{ij}.S_j$. Applying the subtyping relation, we have $S :> AS$, from which Theorem 4 could be inferred.

Theorem 4. *An elementary cycle of length* k *in the initial scheduling matrix is a sequence* $c_i = a_{i1,i2}.a_{i2,i3} \ldots a_{ik,i1}$*, where* $a_{ip,i(p+1) mod\ k} \neq X$ *for all* $p \in \{1..., k\}$*. The throughput of the scheduled system is* $\min_i(|c_i|_1/|c_i|)$ *where* i *ranges over all elementary cycles in the initial scheduling matrix.*

This is obvious because such an elementary cycle c_i is related to an elementary cycle in the original graph, $|c_i|_1$ is equivalent to the number of tokens on c_i and $|c_i|$ is equivalent to the latencies. The throughput obtained from Theorem 4 gives the maximum throughput a scheduled system can achieve. With this information, the periodic calculus can provide a neat representation of the scheduling constraints for this maximum throughput. In following discussion, we only consider the periodic phase of schedulings sequences since the periodic steady state of a system is more important to analyze. Denote the periodic phase of S_i as U_i i.e. $S_i = U_i.U_i \ldots = (U_i)$, we use (U_i) to express the unlimited repetition of U_i. Suppose the throughput obtained from Theorem 4 is $\delta = K_u/T_u$, then $|U_i|_1 = K_u$ and $|U_i| = T_u$ denoting that there are K_u “1” in U_i and U_i is a T_u length sequence. Suppose the initial token of p_i given at the start of the periodic phase is $M_0^u(p_i)$ and the initial scheduling matrix A^u for the periodic phase becomes:

$$a_{ij}^u = \begin{cases} M_0^u(p_j) & t_j \in Pred(t_i) \\ X & otherwise \end{cases} \tag{5.11}$$

Applying the subtyping rule to Equation (5.10) and replacing the scheduling sequences by their periodic phases gives:

$$(U) :> A^u(U), U = (U_1, U_2, \ldots, U_N)^T \tag{5.12}$$

Finding the schedulings is equivalent to finding $[U_i]_p \in \{1, 2, \ldots, T_u\}$, where $p \in \{1, 2, \ldots, K_u\}$, which satisfy Inequality (5.12). Consider any place p_{ij}, let t_j be the transition that consumes token from p_{ij} and t_i be the transition that produce token to p_{ij}. U_i and U_j are the periodic phases of scheduling sequences for t_i and t_j. From (5.12), we have:

$$(U_j) :> M_0^u(p_{ij}).(U_i) \tag{5.13}$$

By Equation (2.3), we have:

$$[(U_j)]_k \geq [(U_i)]_{k-M_0^u(p_{ij})} + 1, k \in \{1,2,3\ldots\} \tag{5.14}$$

Consider the periodicity of (U_j) and apply (5.14) to one clock period, we can infer:

$$\begin{gathered} [U_j]_{(p\ mod\ K_u)} \geq ([U_i]_{(p-M_0^u(pij))\ mod\ K_u} + 1)\ mod\ T_u \\ p \in \{1,2,\ldots,K_u\}, [U_j]_p \in \{1,2,\ldots,T_u\} \end{gathered} \tag{5.15}$$

There are $|P| * K_u$ inequalities in (5.15), where $|P|$ is the number of places. (5.15) gives the full constraints of the schedulings based on which Integer Linear Programming (ILP) can be used for other optimization issues related to the values of $[U_j]_p$. For example, in [39] a buffer size optimization problem with a given throughput for synchronous data flow graph (SDFG) is solved by ILP and the constraints of which can be neatly described by (5.15). Denote B_{ij} as the buffer size of p_{ij}, then the ILP is:

maximize: $\sum B_{ij}$
subject to:
$T_u * B_{ij} + K_u * ([U_i]_p - [U_j]_{p'}) \geq C(p,p'),$
$[U_j]_{(p\ mod\ K_u)} \geq ([U_i]_{(p-M_0^u(pij))\ mod\ K_u} + 1)\ mod\ T_u$
where
$C(p,p') = T_u * (p + 1 - p' + M_0^u(p_{ij}),$
$p, p' \in \{1,2,\ldots,K_u\},$
$[U_j]_p \in \{1,2,\ldots,T_u\}$

We are not going to explain in detail since it is beyond the scope of this chapter, one can refer to [39] for more illustration. Once the scheduling sequences are computed, IP modules as well as relay stations which are modeled by transitions are clock gated by these scheduling sequences. However, tokens may aggregate before join transitions and the fractional synchronizers are proposed to equalize data flow with different rates. In the next section, we discuss how to determine the number of required fractional synchronizers between any two transitions and their enabling sequences based on the schedulings of transitions.

5.2 Fractional synchronizer synthesis

5.2.1 Equalization process

From the previous discussion about the scheduling process, each elementary cycle must work at the same throughput. Therefore, the data flow from the faster cycle should be buffered before join transition, e.g., $T0$ in Fig. 5.2, to match the data flow from the slower cycle. This process is called *equalization* in both [20] and [12]. Simply inserting registers at the faster cycle may not completely equalize their throughputs, and this process may affect the performance of the system. For example in Fig. 5.2, the throughputs of the two elementary cycles $C1$, $C2$ are $1/2$ and $2/5$, and the throughput of the system is $2/5$. However, adding one register to the faster path $C1$ will decrease its throughput to $1/3$ which is less than the system throughput $2/5$. The fact is that since the throughputs are fractional numbers, we need to add fractional latencies to equalize the two cycles. However, adding registers to $C1$ can only add integer latencies. To solve this problem, [20]

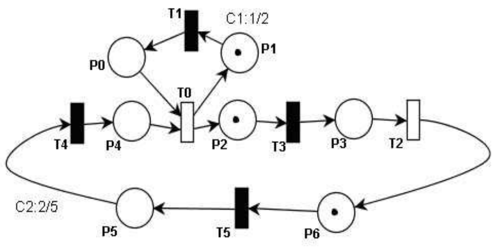


Figure 5.2: A system having two elementary cycles with different throughputs

proposes the fractional synchronizer (Fig. 5.3) and [12] proposes the fractional register. Both of them introduce selective enabled registers to store the aggregated tokens which result in selective delay rather than a constant delay. It will provide better throughput while solving the equalization problem. For example, in Fig. 5.3 the input data will be buffered in the register, thus causing one clock cycle latency of the data flow, only when "1" of the MUX is selected. So one can calculate the activating sequence of the fractional synchronizer without changing the system's throughput. In the following discussion, we focus on insertion of fractional synchronizers and show that the number of such synchronizers and their activation schedulings can be obtained by applying the periodic clock calculus.

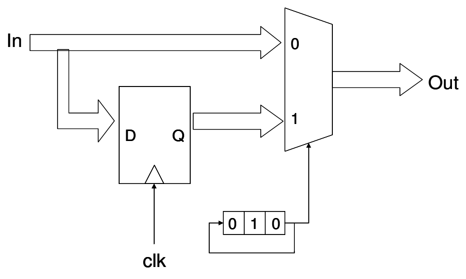


Figure 5.3: Fractional synchronizer ([20])

5.2.2 Fractional synchronizer synthesis

Theorem 5. *[28] If two data flows $f1$ and $f2$ have periodic clocks $clk1$ and $clk2$ respectively which are synchronizable, then the buffer size to synchronize $f2$ to $f1$ is:*

$$\begin{gathered} B(clk2, clk1) = max_k\{|clk2[1 \dots k]|_1 - |clk1[1 \dots k]|_1, 0\}, \\ k \in \{1, 2, 3, \dots\} \end{gathered} \tag{5.16}$$

This basically says that the required buffer size is the maximum number of clock cycles through which the two systems are out of synchronization, before they synchronize again. Note that $B(clk2, clk1)$ is computed based on the fact that only the valid data of $f1$ and $f2$ are buffered. In other words, the data is buffered only its clock is "1" in each instant. Notice that $B(clk2, clk1)$ can be calculated by finite steps, since $clk2$ and $clk1$ are both periodic clocks. In fact, we can write $clk1 = v1(u1)$ and $clk2 = v2(u2)$ where $|u1| = |u2|$ and $|v1| = |v2|$, then

$$\begin{aligned} &|clk2[1 \dots (k + |u1| * m)]|_1 - |clk1[1 \dots (k + |u1| * m)]|_1 \\ &= clk2[1 \dots k]|_1 - |clk1[1 \dots k]|_1, \;\; m \in N \end{aligned} \tag{5.17}$$

Therefore, $clk2[1 \dots k]|_1 - |clk1[1 \dots k]|_1$ is also periodic with period of $|u1|$ and its maximum value $B(clk2, clk1)$ can be determined by trying $k \in \{1, 2 \dots, |u1| + |v1|\}$. Normally, we have $B(clk2, clk1) \neq B(clk1, clk2)$. Based on Theorem 5, we can infer the number of fractional synchronizers needed for the equalization process at a join transition and determine their scheduling sequences.

Theorem 6. *In a marked graph, let t be a join transition and let $Pred(t)$ be the predecessor set of t. Let p_i be the place located between t_i and t where $t_i \in Pred(t)$. Let the initial marking of p_i be $M_0(p_i)$ such that $M_0(p_i) \in \{0, 1\}$. Let the scheduling sequence of t_i be s_i. Define:*

$$s_\sqcup = (\bigsqcup\nolimits_{t_i \in Pred(t)} M_0(p_i).s_i) = s_t \tag{5.18}$$

$$\Delta s_i[k] = |s_i[1...k]|_1 - |s_\sqcup[1...k]|_1 + M_0(p_i) \tag{5.19}$$

Then the number of fractional synchronizers added between p_i and t is:

$$NF_i = max_k(\Delta s_i[k] - 1, 0) \tag{5.20}$$

Let j be the j^{th} fractional synchronizer between p_i and t, then j is scheduled at instant $k+1$, when $j < \Delta s_i[k]$.

We are going to prove that Theorem 6 gives a sufficient condition to make the corresponding marked graph 1-safe.

Proof: Notice that $\Delta s_i[k]$ is the number of tokens which have been produced by t_i while not consumed by t at instant k, if $\Delta s_i[k] > 1$. So we need at least $\Delta s_i[k]$ storage to buffer these tokens at instant k. Since p_i can take 1 token, the rest $\Delta s_i[k] - 1$ tokens should be stored in the fractional synchronizers. Therefore, NF_i fractional synchronizers is enough to store the accumulated tokens for any instant. Denote these fractional synchronizers located from p_i to t as $fs_1, fs_2, \ldots, fs_{NF_i}$. Notice that if a token is stored in one fractional synchronizer at instant k, it will be consumed by a later stage at instant $k+1$ and the control signal of this fractional synchronizer should be enabled at instant $k+1$. This is a fact based on the structure of the fractional synchronizer(see Fig. 7). Since at instant k, $\Delta s_i[k] - 1$ fractional synchronizers are used to store redundant tokens, denoted as $fs_1, fs_2, \ldots, fs_{(\Delta s_i[k]-1)}$, they should be enabled at instant $k+1$. Therefore, the j^{th} fractional synchronizer $fs_j$is enabled at instant $k+1$ if $j \leq \Delta s_i[k] - 1$,i.e., $j < \Delta s_i[k]$. $\square$

Now, we explain this theorem with an example shown in Fig. 5.4 [12] with modifications. Fig. 5.4

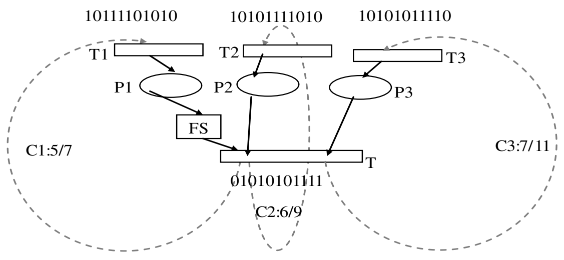


Figure 5.4: A system with three elementary cycles with different throughputs (from [12])

is system with three elementary cycles which have different throughputs. In fact, $\delta(C1) = 5/7$, $\delta(C2) = 6/9$ and $\delta(C3) = 7/11$. T is a join transition with $Pred(T) = T1, T2, T3$. The scheduling sequence $S1$, $S2$ and $S3$ of $T1$, $T2$ and $T3$ are presented in this figure. Initial markings for $P1$, $P2$ and $P3$ are $M_0(P1) = M_0(P2) = M_0(P3) = 0$. The number of fractional synchronizers is computed as shown in Table 5.1. So $NF_1 = max_k(\Delta s_{1,2}[k] - 1, 0) = 1$,

Table 5.1: Computation for the number of fractional synchronizers

	1	2	3	4	5	6	7	8	9	10	11
$S1$	1	0	1	1	1	1	0	1	0	1	0
$S2$	1	0	1	0	1	1	1	1	0	1	0
$S3$	1	0	1	0	1	0	1	1	1	1	0
Δs_1	0	0	0	1	1	2	1	1	0	0	0
Δs_2	0	0	0	0	0	1	1	1	0	0	0
Δs_3	0	0	0	0	0	0	0	0	0	0	0

$NF_2 = max_k(\Delta s_2[k] - 1, 0) = 0$, $NF_3 = max_k(\Delta s_3[k] - 1, 0) = 0$, so only one fractional synchronizer needs to be inserted between $P1$ and T and it should be fired at instant $k + 1$when $\Delta s_1[k] > 1$. i.e. the 7th instant for one clock period. Therefore, its scheduling sequence is $\langle 00000010000 \rangle$.

5.3 Estimation of fractional synchronizer numbers

In Section 5.2.2, we have derived the static computation of fractional synchronizers based on schedulings of a system. Since the MUXs in fractional synchronizers will increase the length of combinational path, placing large number of fractional synchronizers in series should be avoided. However, such information is unclear until the computation of schedulings is complete and which is both time and storage consuming, especially for a large design. In this section, we target this problem by deriving an estimation for the upper bound of fractional synchronizer numbers based only on the structure of a system and which is fast and accurate.

In our estimation, we follow a preprocess from [12] to make the system more close to "equalize". The basic idea is to slow down the faster elementary cycle with integer-cycle latencies by adding relay stations before fractional synchronizers are used. Consider a system containing two elementary circle $C1$ and $C2$, whose mean cycles are $3/4$ and $4/7$ ($3/4 > 4/7$). Thus $C2$ is the critical cycle of the system and the throughput is $4/7$. Instead of computing the scheduling of the system directly, one additional relay station can be added to "slow down" $C1$. After that, the mean cycle of $C1$ is reduced to $3/5$, closer to but still larger than $4/7$. Now, $C1$ is slowed down without deteriorating the system's throughput. Intuitively it can reduce the number of fractional synchronizers. We follow this preprocess by making the assumption in (5.22) and which is discussed later.

Let's consider a system has the throughput of K_u/T_u, C is an elementary cycle of the system and whose mean cycle is N/M $(N/M \geq K_u/T_u)$, where M is the number of transitions of C.

Theorem 7. *Let (U) be the periodic scheduling of one transition t of C. Then: $([U]_{(p+N) \ mod \ K_u} - [U]_p) \ mod \ T_u \geq M$.*

Proof: Since mean cycle of C is the ratio of the number of tokens and the number of transitions on C, N is the number of tokens on C. It takes at least M clock cycles for each token on C to pass through all the transitions on C and therefore within M clock cycles no token can pass one transition t twice, which means t can fire at most N times for every M clock cycles. In other words, the time interval between p'th firing of t and the $(p+N)$'th firing of t should be at least M clock cycles. This proves Theorem 7. $\square$

Theorem 7 also works for N' and M', where $N' = kN$, $M' = kM$ and k is a positive integer. In fact,

$$\begin{array}{c} ([U]_{(p+N') \ mod \ K_u} - [U]_p) \ mod \ T_u = \\ ([U]_{(p+kN) \ mod \ K_u} - [U]_p) \ mod \ T_u = \\ \sum_{j=1}^{k} ([U]_{(p+jN) \ mod \ K_u} - [U]_{(p+(j-1)N) \ mod \ K_u}) \ mod \ T_u \\ \geq \sum_{j=1}^{k} M = kM = M'. \end{array} \tag{5.21}$$

This theorem shows that any segment of U with length M contains at most N "1". For later discussion, we can assume $M < T_u < 2M$ without lose of generality. If $M > T_u$, we can always multiply T_u, K_u with a positive integer to make $M \leq T_u < 2M$, if $T_u > 2M$ we can multiply M,

N with a positive integer to make $T_u < 2M$. They are equivalent to unroll U for some periods to make the value M and T_u close. Another assumption we are going to make is:

$$\frac{N}{M+1} < \frac{K_u}{T_u} \leq \frac{N}{M} \tag{5.22}$$

This assumption is actually the result after the preprocessing [12]. The right part of (5.22) is true by definition. In case the left part does not hold, i.e.,$\frac{N}{M+1} > \frac{K_u}{T_u}$, since M is the number of transitions (latencies) in C, we can add one more transition (latency) as a relay station to C making $M_1 = M + 1$, if still $\frac{N}{M_1+1} > \frac{K_u}{T_u}$, we can add another transition (latency) on C to make $M_2 = M_1 + 1$. This process can continue until it reaches M' such that $\frac{N}{M'+1} < \frac{K_u}{T_u} \leq \frac{N}{M'}$. Also, since $\frac{K_u}{T_u} \leq \frac{N}{M'}$, the throughput of the system does not change. This process is equivalent to slowing down C and making N/M and K_u/T_u as close as possible without deteriorating the system's throughput. With these assumptions, we could infer that:

$$\begin{cases} N < K_u(\frac{M+1}{T}) \leq K_u \;\; if\ M < T \\ N = K_u \qquad\qquad\quad if\ M = T \end{cases} \Rightarrow N \leq K_u \tag{5.23}$$

Theorem 8. *Let $U = u_1.u_2 \dots u_{M-1}.u_M \dots u_{T_u}$ and define $U(u_i)_M = u_i.u_{i+1} \dots u_{(i+M-1)\ mod\ T_u}$ be a segment of U with length M and starting from u_i, where $i \in \{1, 2 \dots T_u\}$. Then $\exists i \in \{1, 2, \dots, T_u\}$ that $|U(u_i)_M|_1 = N$.*

Proof: We are going to prove it by contradiction. Suppose for all $U(u_i)_M$, $|U(u_i)_M|_1 < N$, indicating that the time interval between the p'th 1 of U and the $(p + N - 1)$'th 1 of U should be at least M clock cycles,i.e.,

$$\begin{cases} [U]_N - [U]_1 \geq M \\ [U]_{N+1} - [U]_2 \geq M \\ \dots \\ [U]_{K_u} - [U]_{K_u-N+2} \geq M \\ [U]_1 + T_u - [U]_{K_u-N+1} \geq M \\ [U]_2 + T_u - [U]_{K_u-N} \geq M \\ \dots \\ [U]_{N-1} + T_u - [U]_{K_u} \geq M \end{cases} \tag{5.24}$$

Summing these inequalities up will lead to:

$$T_u * (N-1) \geq K_u * M \Rightarrow \frac{N-1}{M} \geq \frac{K_u}{T_u} \tag{5.25}$$

On the other hand,

$$\frac{N-1}{M} - \frac{N}{M+1} = \frac{N-1-M}{M*(M+1)} < 0 \tag{5.26}$$

Combined with (5.22) leads to:

$$\frac{N-1}{M} < \frac{N}{M+1} < \frac{K_u}{T_u} \tag{5.27}$$

which contradicts (5.25). Therefore Theorem 8 is proved. □

Let's consider a situation when C has only one join transition t_j, transition t is also in C and $t \in Pred(t_j)$. Let the scheduling of t and t_j be (U) and (U_j), since we are talking about ASAP scheduling, U is obtained by right rotating U_j for $M-1$ bits. $[U_j]_p$ and $[U]_p$ can be related by Equation (5.28):

$$[U_j]_p = ([U]_{p+N-u_M \; mod \; K_u} - M + 1) \; mod \; T_u \tag{5.28}$$

Let $U = u_1.u_2 \dots u_{M-1}.u_M \dots u_{T_u}$, then $U_j = u_M \dots u_{T_u}.u_1.u_2 \dots u_{M-1}$. By Theorem 7, there is a segment $U(u_i)_M$ that $|U(u_i)_M|_1 = N$. Since U is a periodic sequence, we can assume that $|U(u_1)_M|_1 = N$ without losing generality. We are going to explain Equation (5.28). In $u_1.u_2 \dots u_{M-1}$, there are $N - |u_M|$ "1" ($|u_M| = 1$ if $u_M =$"1", else $|u_M| = 0$) so the p'th "1" in U_j corresponds to the $p + (N - |u_M|)$'th "1" in U. In addition, the distance between U_j and U for the same u_k is $M-1$ and which gives Equation (5.28).

Since t_j is the only reconvergent transition in C, tokens can only aggregate before t_j by ASAP scheduling. Let the place between t and t_j be p, let B be the maximum number of tokens that aggregate in p, then $\exists p0 \in \{1, 2, \dots, K_u\}$ that $[U]_{p0+B} < [U_j]_{p0}$ and $[U]_{p0+B+1} \geq [U_j]_{p0}$. It means that the $(p0 + B)$'th firing of t happens before the p'th firing of U_j and therefore B tokens are aggregated on p and p's size should be at least B. Recall that this discussion does not consider the initial token on p and we deal with that later. Using (5.28), we have:

$$[U]_{p0} \leq \; ([U]_{p0+N-|u_M|-B \; mod \; K_u} - M + 1) \; mod \; T_u \tag{5.29}$$

This relation means that there is a segment in U that:

$$\begin{gathered} |U(u_k)_{M-1}|_1 = N - |u_M| - B + 1 = N - B', \\ B' = B + |u_M| - 1 \end{gathered} \tag{5.30}$$

Now the problem becomes how large the value B' can be without breaking the constraints shown in Theorem 7 and Theorem 8. We first consider how large the value B' can be in a M length sequence since the difference between M long segment and $M-1$ long segment is no more than 1.

Theorem 9. *For any segments $U(u_i)_M$ and $U(u_{i+1})_M$, $||U(u_i)_M|_1 - |U(u_{i+1})_M|_1| \leq 1$*

Proof: Recall that $U(u_i)_M = u_i.u_{i+1} \dots u_{(i+M-1) \; mod \; T_u}$ and $U(u_{i+1})_M = u_{i+1}.u_{i+2} \dots u_{(i+M) \; mod \; T_u}$. There are only one bit difference between $U(u_i)_M$ and $U(u_{i+1})_M$. The difference for number of "1"s in $U(u_i)_M$ and $U(u_{i+1})_M$ depends only on $|u_i - u_{(i+M) \; mod \; T_u}|$ and which is no more than 1. It proves Theorem 9. □

Now, we are ready to give a upper bound on B'.

Theorem 10. *One upper bound of B' is $(T_uN - K_uM)^{1/2}$.*

Proof: Let sequence $U(u_{i'_B})_M$ have the least number of "1"s and which is $N - B'$, i.e.,

$$|U(u_{i'_B})_M|_1 = N - B', B' \leq N \tag{5.31}$$

By Theorem 9,

$$\begin{array}{c} |U(u_{(i'_B+1)\ mod\ T_u})_M|_1 \leq N - B' + 1 \\ |U(u_{(i'_B-1)\ mod\ T_u})_M|_1 \leq N - B' + 1 \end{array} \tag{5.32}$$

Further we can infer that,

$$\begin{array}{c} |U(u_{(i'_B+2)\ mod\ T_u})_M|_1 \leq N - B' + 2 \\ |U(u_{(i'_B-2)\ mod\ T_u})_M|_1 \leq N - B' + 2 \end{array} \tag{5.33}$$

By Theorem 8, $\exists k \in \{1, 2, \ldots, T_u\}$ that $|U(u_k)_M|_1 = N$ and if continue deriving such inequalities, the upper bound of these inequalities will finally reach N,i.e.,

$$\begin{array}{c} \ldots, \ldots \\ |U(u_{(i'_B+B'-1)\ mod\ T_u})_M|_1 \leq N - 1 \\ |U(u_{(i'_B-B'-1)\ mod\ T_u})_M|_1 \leq N - 1 \end{array} \tag{5.34}$$

$$\begin{array}{c} \forall\ k \notin \{(i'_B - B' - 1)\ mod\ T_u, \ldots, (i'_B + B' - 1)\ mod\ T_u\}, \\ |U(u_k)_M|_1 \leq N \end{array} \tag{5.35}$$

It implies that $\{(i'_B - B' - 1)\ mod\ T_u, \ldots, (i'_B + B' - 1)\ mod\ T_u\}$ must be a subset of $\{1, 2, \ldots, T_u\}$, assume there are x inequalities in (5.35), then:

$$\begin{array}{c} x = T_u - (i'_B + B' - 1 - (i'_B - B' - 1) + 1), x \geq 1 \\ \Rightarrow x = T_u - 2B' + 1, x \geq 1 \end{array} \tag{5.36}$$

Summing up (5.31) to (5.35):

$$\begin{array}{l} \sum_{i=1}^{T_u} |U(u_i)_M|_1 \leq N - B' + 2\sum_{j=1}^{B'-1} (N - j) + xN \\ = N - B' + (B' - 1)(2N - B') + xN \\ = -B'^2 + 2NB' + (x - 1)N \end{array} \tag{5.37}$$

On the other hand, the sum of $|U(u_i)_M|_1$ counts each "1" in U for M times and which indicates:

$$\sum_{i=1}^{T_u} |U(u_i)_M|_1 = MK_u \tag{5.38}$$

Take (5.38) and (5.36) into (5.37) and with some rearrangement we have:

$$B'^2 \leq NT_u - MK_u \Leftrightarrow B' \leq (NT_u - MK_u)^{1/2} \tag{5.39}$$

This proves Theorem 10. □

Let $B^* = \lfloor (NT_u - MK_u)^{1/2} \rfloor$ and in later discussion we use B^* to represent the upper bound of B' given by Theorem 10. From previous discussion, $N - B^*$ become a lower bound of $|U(u_i)_M|_1$ which is also a lower bound of $|U(u_i)_{M-1}|_1$. Take this into (5.30):

$$\begin{gathered} |U(u_k)_{M-1}|_1 = N - |u_M| - B + 1 \geq B^* \\ \Rightarrow B \leq B^* - 1 + |u_M| \end{gathered} \tag{5.40}$$

Recall that B is the maximum number of tokens which can be aggregated on place p, by Theorem 7 the number of fractional synchronizers needed before p is $B - 1 + M_0(p)$. So we have:

Theorem 11. *Given a system whose throughput is K_u/T_u, C is one elementary cycle of the system whose mean cycle is N/M, t is the only join transition in C and p is the place before t. Then the number of factional synchronizers NF is upper bounded at place p by $FR^* = B^* - 2 + |u_M| + M_0(p)$, where $B^* = (NT_u - MK_u)^{1/2}$.*

In this theorem $|u_M|$ is unknown before the scheduling is computed, so with a conservative approximation, we should replace it by "1" in the theorem.

5.4 Experimental Results

5.4.1 Computation of fractional synchronizer

Experimental results of the fractional synchronizer computation based on Theorem 6 are shown in Table 5.2. In the experiment, TEST0 $\sim$ TEST8 are randomly generated marked graph models. Latency is randomly generated on each link to simulate the wiring latency after floor planning. Relay stations are added to make the latency of each link less than unit time. We calculate the throughputs and the numbers of fractional synchronizers. We also use ISCAS'89 benchmark to test our theorems and in this case, each gate is modeled as a transition in the marked graph model. Even though in most cases latency-insensitive protocol will not be implemented in such a fine granularity, however, it is reasonable to use the topology of these benchmarks to test our method. The experiment is carried out on a PC with Genuine Intel CPU T1300 at 1.66GHz with 1G memory. The results are shown in Table 5.2.

5.4.2 Evaluation of the upper bound for fractional synchronizer numbers

We also evaluate the upper bound obtained in Theorem 11. Notice that in the discussion of Section 5.3, we didn't consider the initial token distribution of the system. Therefore, the upper bound we derived is for the worst case distribution. That is the initial token distribution which could result in the largest number of fractional synchronizers. Therefore, we should compare this upper bound with the worst case distribution. However, it is quite difficult to enumerate all the initial tokens to get the worst case. Instead we construct the scheduling sequence U that leads to the largest number of fractional synchronizers based on the system's structure and make the comparison based on this U. By (5.30), $\min|U(u_k)_{M-1}|_1$,i.e., the minimum numbers of "1" contained in any $M-1$ long sequences should be obtained. This can be easily achieved by Integer Linear Programming (ILP). In later discussion, N, M, K_u, T_u have the same symbolic meanings as in Section 5.3. Let NR_k be the numbers of "1" contained in sequence $U[1,\dots,k]$, $k \in \{1,2,\dots,T_u\}$. Now, the ILP can be formulated:

$$\begin{aligned}
&\text{minimize: } NR_{M-1}\\
&\text{subject to:}\\
&\quad NR_{M+k} - NR_k \le N,\ k \in \{1,\dots,Tu-M\}\\
&\quad K_u + NR_k - NR_{Tu-M+k} \le N,\ k \in \{1,\dots,M\}\\
&\quad 0 \le NR_{k+1} - NR_k \le 1,\ k \in \{1,\dots,Tu-1\} && \text{(a)}\\
&\quad 0 \le NR_1 - NR_{T_u} + K_u \le 1 && \text{(b)}\\
&\quad NR_{T_u} = K_u && \text{(c)}
\end{aligned}$$

Constraints (a) basically regulate that any M length sequences of U should have at most N "1"s which is required by Theorem 7. Constraints (b) is from Theorem 9. $NR_{T_u} = |U|_1 = K_u$ is the constraint (c). Compared to Theorem 10, this ILP gives the exact value of $\min|U(u_i)_{M-1}|_1$ and $N - \min|U(u_i)_{M-1}|_1$ gives the exact upper bound of B' in Theorem 9. However, remember that solving ILP is NP-complete and it is only used here to construct a test pattern for comparison and which should be avoided in the analysis for real design. That is the reason why we have come up with the estimated upper bound which can be efficiently computed. On the other hand, $U = u_1.u_2\dots u_{T_u}$ can be computed as a byproduct of this ILP as $u_i = NR_i - NR_{i-1}$, if $i > 1$ and $u_1 = NR_1$.

Now, we are going to construct the tested marked graph and its initial token distribution. Since in Section 5.3, we focus on an elementary cycle which have only one join transition, the marked graph generated for testing has only two elementary cycles $C1$ and $C2$ among which only one transition t_j is shared. $C1$ has N initial tokens and M transition, $C2$ has K_u initial token and T_u transitions. So the mean cycle of $C1$ and $C2$ are N/M and K_u/T_u ($N/M > K_u/T_u$). Name the places in $C1$ by this rule, the place connected to the outgoing arc of t_j is named as $p_{1,1}$, and the place connected to the incoming arc of t_j is named as $p_{1,M}$, the places between these two are named as $p_{1,2}$ to $p_{1,M-1}$. Places in $C2$ are named by the same rule as $p_{2,1}$ to p_{2,T_u}. The initial token of $C1$ and $C2$ are deployed as:
1) For $C1$, find a sequence $U(u_i)_M$ that $|U(u_i)_M|_1 = N$, rotate U to get $U' = u'_1.u'_2\dots u'_{T_u}$ and make $u'_1 = u_i$. Put an initial token on $p_{1,k}$, $k = 1,2,\dots,M$, iff $u_{M-k'} =$"1".
2) For $C2$, left-rotate U' for $M-1$ bits to get $U'' = u''_1.u''_2\dots u''_{T_u}$. Put an initial token on $p_{2,k}$, $k = 1,2,\dots,T_u$, iff $u_{T_u-k+1} =$"1".

The tested marked graph is generated according to the discussion above, the ASAP scheduling of the this marked graph is computed and the number of fractional synchronizers FR is obtained by

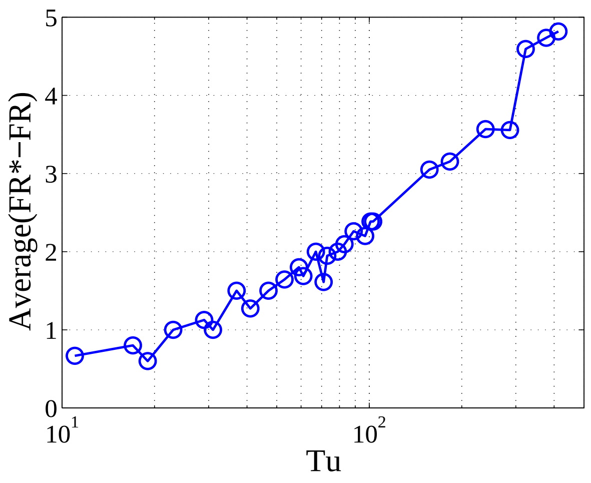


Figure 5.5: Comparison of fractional synchronizer numbers between FR and FR^* at different throughput.

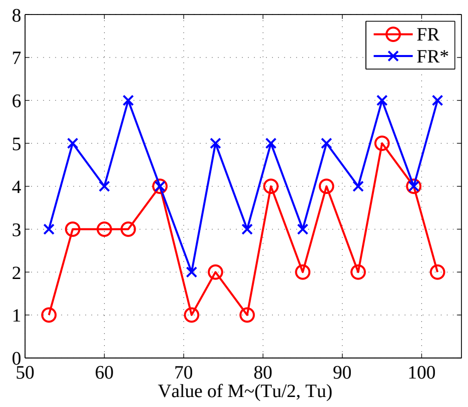
Theorem 6. On the other hand, the upper bound FR^* we derived in Section 5.3 could be computed according to Theorem 11. The comparison between FR and FR^* is carried out to evaluate the accuracy of FR^*. Note that the way we generate the tested marked graph may not result in the worst case of FR such that FR is maximized, however it does not hurt the evaluation. Because, $FR^* - FR$ is always larger than the worst case.

A group of prime numbers ranging in $10 \sim 500$ are selected as T_u. K_u is chosen as $(T_u - 1)/2$. This is to make $gcd(K_u, T_u) = 1$. For each (K_u, T_u), we compute the average value of $(FR^* - FR)$ through different M, where M iterates from $\lceil T_u/2 \rceil$ to $T_u - 1$. According to (5.22), we can infer that given M, K_u, T_u, either N is $\lceil M * K_u/T_u \rceil$ or N does not exist. (In fact, the range of N is $(M+1) * K_u/T_u - M * K_u/T_u = K_u/T_u < 1$ and within which no more than 1 positive integer exists. If such integer exists, it must be $\lceil M * K_u/T_u \rceil$.) $\overline{(FR^* - FR)}$ are calculated based on the valid pair (if N exists) (N, M, K_u, T_u) and the result is shown in Fig. 5.5. The x-axis represents the range of T_u and the y-axis represents the according $\overline{(FR^* - FR)}$. X-axis uses logarithmic scale, because more samples of T_u are chosen from (10,100). Normally, $\overline{(FR^* - FR)}$ increases as T_u increase but $\overline{(FR^* - FR)} < 5$ if $T_u < 500$. When $T_u < 100$, $\overline{(FR^* - FR)} < 3$ and mostly $\overline{(FR^* - FR)} < 2$. Since T_u is the number of transitions in an elementary cycle and in real design it presents the number of IP modules in an elementary cycle, T_u is generally less than 100 for current SOC design. With this consideration, our upper bound FR^* is quite accurate.

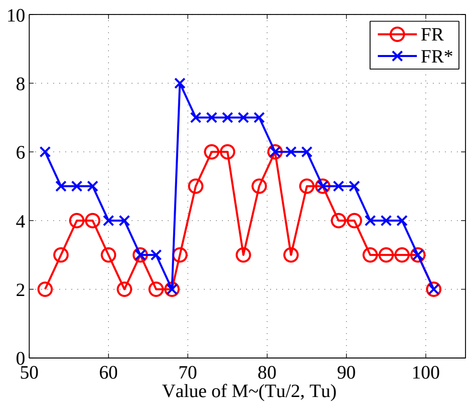
On the other hand, the maximum of $FR^* - FR$ may be *filtered* in $\overline{(FR^* - FR)}$ thus only evaluating $\overline{(FR^* - FR)}$ can not subjectively reflect the real accuracy. In addition, whether the accuracy is affected by the value of throughput under the same T_u deserves investigation and which is helpful to find a better estimation. Another is carried out to complement our experiment. $T_u = 103$ is selected because it is large enough to reflect the difference and represents the upper bound of SOC size. Four different pairs of (K_{u_i}, T_u) $(i = 1, 2, 3, 4)$ are used to assess FR^* at four distinct throughput. For each (K_{u_i}, T_u), M is selected from $\lceil T_u/2 \rceil$ to $T_u - 1$, N is determined as the same manner of above discussion and FR and FR^* are computed and compared for every valid (M, N) pair and the results are shown in Fig. 5.6$(a) \sim (d)$. For (K_u, T_u) pair $(29, 103)$, $(53, 103)$, $(79, 103)$, FR^* matches FR quite well while for pair $(97, 103)$, FR^* deviates a little more from FR. As demonstrated by the results, estimation FR^* becomes less accurate when throughput K_u/T_u is close to 1. One possible reason could be that when K_u is close to T_u, there are few number of "0"s inside T_u. It might prevent more Inequalities from (5.31) to (5.35) achieving the equality. Therefore, the upper bound becomes less tight.

Table 5.2: Experimental results of fractional synchronizer number computation

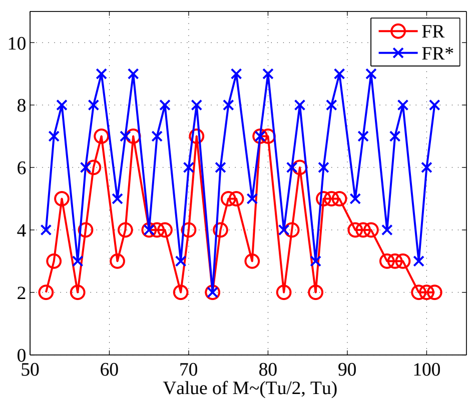
Bench	LIS Info.					Throughput	Performance	
mark	#Trans.	#Places	#RS	#Join Trans.	#Frac. sync.	δ	Exec. time(s)	Mem.(M)
test0	9	12	4	3	3	0.43	0.03	< 2
test1	17	26	11	10	6	0.20	0.04	< 2
test2	17	18	9	2	3	0.47	0.05	< 2
test6	18	20	10	3	4	0.20	0.03	< 2
test7	26	43	17	15	20	0.17	0.23	< 2
test8	10	14	5	4	4	0.33	0.06	< 2
test9	16	23	10	8	6	0.62	0.14	< 2
s27	18	29	11	12	8	0.60	0.11	2.95
s382	188	348	170	112	90	0.50	1.16	3.76
s444	211	393	196	131	75	0.50	1.17	3.77
s1423	753	1311	632	542	407	0.50	3.86	15.14
s5378	3042	4641	2293	1170	1686	0.53	128.16	105.72



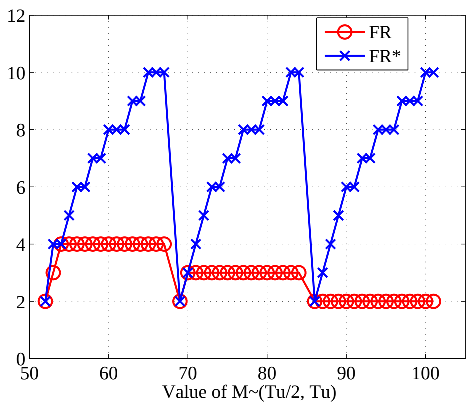
(a) $K_u = 29, T_u = 103$



(b) $K_u = 53, T_u = 103$



(c) $K_u = 79, T_u = 103$



(d) $K_u = 97, T_u = 103$

Figure 5.6: Comparison of fractional synchronizer numbers between FR and FR^* at different throughput ($T_u = 103$)

Chapter 6

Minimizing Back Pressure for LIS Synthesis

Most scheduling based latency insensitive designs in the literature focus on systems whose graphical representation is a single strongly connected component (SCC), where a hand-shake based protocol can be replaced by periodic clock gating through ASAP scheduling. However, for systems that are represented as interconnected SCCs, ‘back pressure’, always implemented as the ‘stall’ signal in the backward directions between SCCs, is required to prevent overflow. In this chapter, we formulate the problem of finding a minimum set of back pressure edges. We show that this problem can be reduced to the Minimum Cost Arborescence (MCA) problem for directed graphs. This allows us to obtain a polynomial time algorithm for synthesizing a minimum cost latency insensitive implementation starting from a synchronous model of the original system. We also show that implementing back pressure edges for every inter-SCC connection, as done in a regular hand-shake based protocol, is inferior for the overall system’s throughput. This chapter provides a formal framework for converting a synchronous model into a latency insensitive implementation with a minimum number of inter-SCC back pressure edges and for leveraging periodic clock based scheduling of intra-SCC latency insensitivity.

6.1 Model LIS using partially back pressure graph

We hope to optimize an LIS by removing unnecessary back pressure arcs. A model between MG and EMG is required to analyze an LIS where only some of the interconnects have back pressure. In this section, we propose *partial back pressure graph* (PBPG), an extension of MG model which classifies arcs A into three categories A_U, A_F and A_B.

Definition 12 (PBPG)**.** *A partial back pressure graph (PBPG) is a 4-tuple* (T, A, M_0, B)*, such that:*

- $\langle T, A, M_0 \rangle$ *is an Marked Graph;*

- *A consists of three subsets A_U, A_F and A_B and is associated with a labeling function $L : A \rightarrow \{F, B\}$:*
  - *A_U: set of forward arcs without back pressure arcs, $\forall a \in A_U$, $L(a) = F$;*
  - *A_F: set of forward arcs with back pressure arcs, $\forall a \in A_F$, $L(a) = F$;*
  - *A_B: set of back pressure arcs, $\forall a \in A_B$, $L(a) = B$;*

  A bijection exists between A_F and A_B that for each arc $a(t_s, t_d) \in A_F$, $\exists a'(t_d, t_s) \in A_B$. $a \in A$ has unit time delay.
- *$B : A_U \cup A_F \rightarrow \mathbb{N}$ is the upper bound on the capacity of forward arcs. $\forall a \in A_U$, $B(a) = \infty$; $\forall a \in A_F$, $B(a) \geq 2$;*

6.1.1 Model LIS with or without back pressure using PBPG

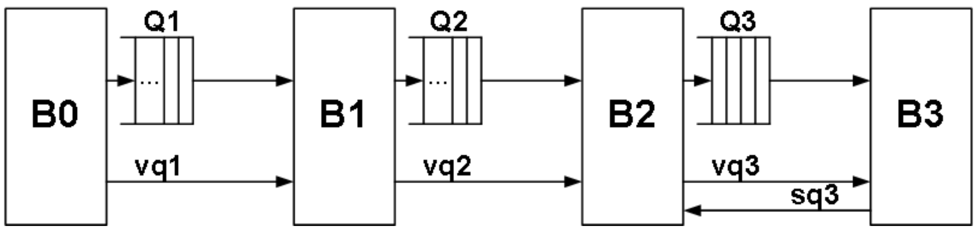
Now, we present how an LIS is modeled by PBPG. We assume that each RS or IP block has one stage pipeline such that it can be regarded as a composition of combinational logics and one stage flip-flops. IPs with multiple pipeline stages can be divided into several sub-blocks in serial. We model an LIS with partial back pressure according to the following rules:

1. The combinational logics of computation block B (either IP modules or RSs) is modeled by transitions t. The storages of B are modeled by arcs $a \in t^\bullet \cap \{A_U \cup A_F\}$
2. For forward arc $a \in A_U \cup A_F$, $M_0(a) = 1$ if a models the storage of an IP module. Otherwise, $M_0(a) = 0$.
3. Arc in $a \in A$ has unit latency by the fact that each block has single pipeline stage. It relates to one clock cycle delay in LIS.
4. A PBPG model is fired with ASAP scheduling. “One clock cycle transaction of a PBPG” means one time ASAP scheduling of the PBPG.
5. $M(a)$ ($a \in A_U \cup A_F$) equals the number of valid data stored in the buffer modeled by a. Arcs in A_B are drawn by dash lines. $a' \in A_B$ has the initial token $M_0(a') = B(a) - M_0(a)$, where a is its corresponding forward arc $p \in A_F$. $M(a')$ represents the number of available buffers for a. $M(a') = 0$ means a is full and IP blocks modeled by ${}^\bullet a$ should be stalled.

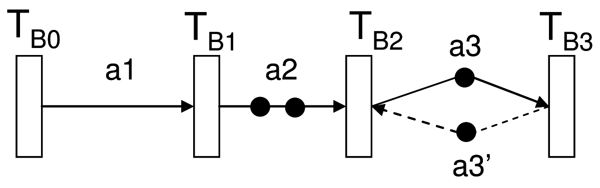
Now, a small example shown in Fig. 6.1(a) and 6.1(b) is used to explain how an LIS is modeled by PBPG. Four blocks $Bi(i = 0, 1, 2, 3)$ are connected in Fig. 6.1(a) . Their combinational logics are modeled as transitions T_{Bi} in Fig. 6.1(b). Bi’s output buffer is Qi packed with signal vqi which indicates whether Qi has valid data. The packed output buffers are modeled as arcs ai. $M(ai)$

represents the number of valid data in Qi. $sq3$ is the "stall signal" from $B3$ to $B2$ and which is modeled by the dashed arc $a3'$. $M(a3') = B(a3) - M(a3)$ represents the number of available buffers on edge $a3$ which is the back pressure arc for $a3$. $M(a3') = 0$ means $a3$ is full. $a1$ and $a2$ have no back pressure, so $a1, a2 \in A_U$. $a3$ has back pressure $a3'$, so $a3 \in A_F$ and $a3' \in A_B$.

Enabling of $B2$ depends on both inputs validity ($Q2$ has valid data) and outputs buffer availability ($Q3$ is not full). While $B1$ is enabled only when $Q1$ has valid data which assumes that $Q2$ can always hold data produced by $B1$. Even though $B1$ and $B2$ are activated in different manners, these activation conditions can be uniformly represented by its PBPG model that a transition is able to fire when all of its incoming arcs are marked. Eg. T_{B2} can fire when both $M(a2) > 0$ and $M(a3') > 0$ while T_{B1} can fire when $M(a1) > 0$. Note that overflow happens on $Q2$ in this example.



(a)



(b)

Figure 6.1: A diagram of an LIS (a), and its PBPG representation (b)

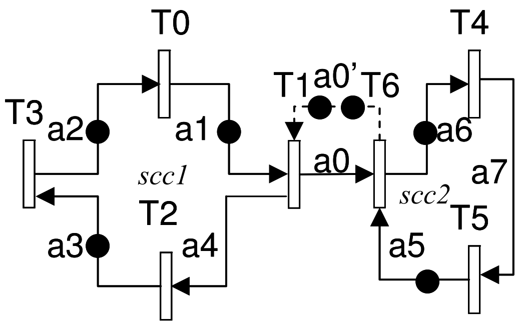
In the discussion of this chapter, we start from a system for which $A_F = A_B = \phi$ (no back pressure arc). Then back pressure arcs are gradually added to arcs $a \in A_U$ which moves a from A_U to A_F. Eventually, only finite number of tokens will reside on arc $a \in A_U$ under ASAP scheduling. More discussion can be found in later sections.

6.1.2 Performance analysis based on PBPG

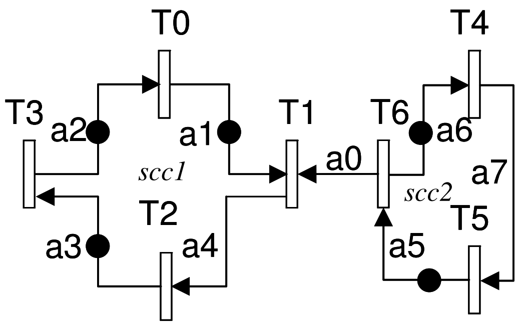
A *directed cycle* C in a PBPG is formed by arcs in A. The *cycle-mean* of C is $\delta(C) = \frac{\sum_{a \in C} M_0(a)}{|C|}$, where $|C|$ is the number of arcs that consists of C and is also the latency of C. The throughput of a SCC scc_k is defined by $\delta(scc_k) = \min_{C \in scc_k} \delta(C)$. For a system G consisting of multiple SCCs, the maximum achievable throughput is $\delta_U = \min_{scc_k \in G} \delta(scc_k)$. The cycle whose cycle-mean equals δ_U is denoted as *critical cycle* C_0.

6.1.3 Interconnections between SCCs

Now, we describe the boundedness problem by considering the interconnections between two SCCs. In Fig. 6.2(a) and Fig. 6.2(b), scc_1 has four transitions $\{T0, T1, T2, T3\}$ and $\delta(scc_1) = 3/4$. scc_2 has three transitions $\{T4, T5, T6\}$ and $\delta(scc_2) = 2/3$. scc_1 and scc_2 are connected through $a0$. If the data flow is from scc_1 to scc_2 (Fig. 6.2(a)), $T1$ produces three tokens into $a0$ for every four clock cycles and $T6$ consumes two tokens from $a0$ for every three clock cycles. This leads to overflow on $a0$. On the other hand, if the data flow is from scc_2 to scc_1 (Fig. 6.2(b)), $T1$ can't fire until $M(a0) > 0$, so $a0$ is bounded. Even though three tokens will aggregate at $a1$ because of barrier synchronization, we can increase the capacity of $a1$ with bounded value for the aggregated tokens and no overflow happens in this situation. This observation leads to Theorem 12.



(a) communication: scc1 to scc2



(b) communication: scc2 to scc1

Figure 6.2: Communication between two SCCs with different throughputs: scc1(3/4), scc2(2/3)

Theorem 12. *Let scc_1 and scc_2 be two SCCs in a PBPG with $\delta(scc_1) > \delta(scc_2)$. Let there be unidirectional links between scc_1 and scc_2 in any direction. The subsystem consisting of scc_1 and scc_2 is bounded with ASAP scheduling if and only if the unidirectional links are from scc_2 to scc_1.*

Proof: By [61] any strongly connected MG is bounded, so is strongly connected PBPG. Therefore, SCC1 and SCC2 are bounded. We only need to focus on the inter-SCC links. "$\Leftarrow$": Suppose the unidirectional links are from scc_1 to scc_2. Let $t_0 \overset{a_0}{\to} t'_0$ be one of these unidirectional links, where t_0 is a transition of scc_1 and t'_0 is a transition of scc_2 and a_0 is the arc between them. The firing of transitions in scc_1 is independent from the firing of transitions in scc_2. Let $\delta(scc_1) = \frac{m1}{n1}$ and $\delta(scc_2) = \frac{m2}{n2}$. Since $\delta(scc_1) > \delta(scc_2)$, we have $(m1 * n2 - m2 * n1) > 0$. Over $n1 * n2$ clock cycles, t_0 produces $m1 * n2$ tokens while t'_0 only consumes $m2 * n1$ tokens. Thus, $(m1 * n2 - m2 * n1) > 0$ tokens will aggregate on arc a_0 during this time interval. Therefore, over time unbounded number of tokens will aggregate on arc a_0; that is, the system becomes unbounded. "$\Rightarrow$": We follow the previous notation but change the link's direction as $t'_0 \overset{a_0}{\to} t_0$. Thus, firing of transitions in scc_1 depends on the firing of transitions in scc_2. t_0 can't fire until there are tokens on a_0, which are produced by t'_0. Since $\delta(scc_1) > \delta(scc_2)$, t_0 can fire no more frequently than t'_0. Thus, tokens will not aggregate on a_0 and the system is bounded. □

In case overflow occurs, back pressure arcs must be added to make system bounded. For example, $a0'$ is added for $a0$ which stalls $T1$ if $a0$ is full. This can also be viewed as making the system strongly connected to achieve boundedness [61]. This fact is also observed in [17]. After $a0'$ is added, the throughput of the new SCC is 2/3 which equals the throughput of scc_2, which is the SCC with smaller throughput. This is not always the fact. In general cases, when two SCC scc_1 and scc_2 are merged into one SCC scc_{12} through added back pressure arcs, we have $\delta(scc_{12}) \leq \min\{\delta(scc_1), \delta(scc_2)\}$. Whether the equality hold depends on the buffer sizes of the inter-SCC arcs and which will be discussed in more detail in Section 8.3. In the following discussion we assume:

Assumption 1. *when several SCCs: scc_1, scc_2, ..., scc_k are merged into one scc_m due to added back pressure arcs, we always have enough buffer sizes for the inter-SCC arcs to make $\delta(scc_m) = \min\{\delta(scc_i), i = 1, 2 \ldots, k$.*

This assumption is not guaranteed in the discussion of back pressure minimization process. We postpone the discussion in Chapter 8. In this chapter, when a back pressure a' is added to A_B, we simply make $B(a) = 2$, where $a \in A_F$ is the corresponding forward arc of a'. So the initial token $M_0(a') = B(a) - M_0(a) = 2 - M_0(a)$.

6.2 Boundedness analysis with strongly connected component graph

In this section, strongly connected component graph (SCCG) is proposed to extend the boundedness analysis for a systems with multiple-SCC, where each SCC has a different throughput. SCCG is constructed from a PBPG model. We only focus on connected systems; thus, we assume that the graphs under consideration have a single connected component when the directions on the edges are removed.

Definition 13 (Strongly Connected Component Graph). *$SCCG(V_{scc}, A_{scc}, \delta, \delta', MinV)$ is a direct acyclic multi-graph constructed from a given PBPG model G.*

- *Each $v \in V_{scc}$ is an SCC of G.*
- *Each arc $a(v1, v2) \in A_{scc}$ connects two SCC $v1$ and $v2$.*
- *$\delta : V_{scc} \to \mathbb{Q}$, $\delta(v)$ is the throughput of v when v is disconnected from other SCCs.*
- *$\delta' : V_{scc} \to \mathbb{Q}$, $\delta'(v)$ is the throughput of v when connected to the entire system. (We require that $\delta'(v) = \delta(v)$ initially, then $\delta'(v)$ changes when connecting to other SCCs.)*

- *For trivial SCC $v \in V_{scc}$, which has a single transition, we assume $\delta(v) = 1$.*
- *$MinV \subseteq V_{scc}$ is the set of SCCs which have the minimum throughput. The maximum throughput G can achieve is $\delta_U = \delta(v)$, where $v \in MinV$. $\delta(MinV)$ is used to represent $\delta(v), v \in MinV$.*

There may be more than one arc connecting two SCCs in an SCCG but these arcs should have the same direction; otherwise, the two SCCs will be merged to form just one SCC. The difference between $\delta(v)$ and $\delta'(v)$ is the following: $\delta(v)$ is the throughput when v is disconnected from other components; therefore, $\delta(v)$ is determined only by v's topology and can be regarded as a *local throughput*. On the other hand, $\delta'(v)$ reflect the throughput change when v is connected to other SCCs which can be regarded as the *global throughput*. A small example of SCCG constructed from a PBPG is shown in Fig. 6.2(b) and Fig. 6.2(a). By Definition 13, $V_{scc} = \{v1, v2, v3, v4\}$, $v1$, $v2$, $v4$ are three trivial SCCs that only contain singular transitions $T0$, $T1$, and $T5$ respectively. $v3$ is consists $\{T2, T3, T4\}$ and their interconnections. $A_{scc} = \{a0, a1, a3, a5\}$. $a1$ and $a3$ have the same direction from $v2$ to $v3$. $\delta(v1) = \delta(v2) = \delta(v4) = 1$ and $\delta(v3) = 1/3$.

We note that SCCs of a directed graph $G(V, E)$ can be found in time $O(|V| + |E|)$ using Tarjan's algorithm [74]. The throughput of each SCC can be calculated as the minimum cycle mean using the algorithm given in [34]. The algorithm has the complexity of $O(m)$ for the best case and $O(mn)$ for the worst case, where m and n are the number of arcs and transitions in one SCC. It is easy to show that the SCCG is weakly connected (i.e., when all of its directed arcs are replaced by undirected arcs, the resulting undirected graph is connected).It is easy to show that the SCCG is weakly connected (i.e., when all of its directed arcs are replaced by undirected arcs, the resulting undirected graph is connected).

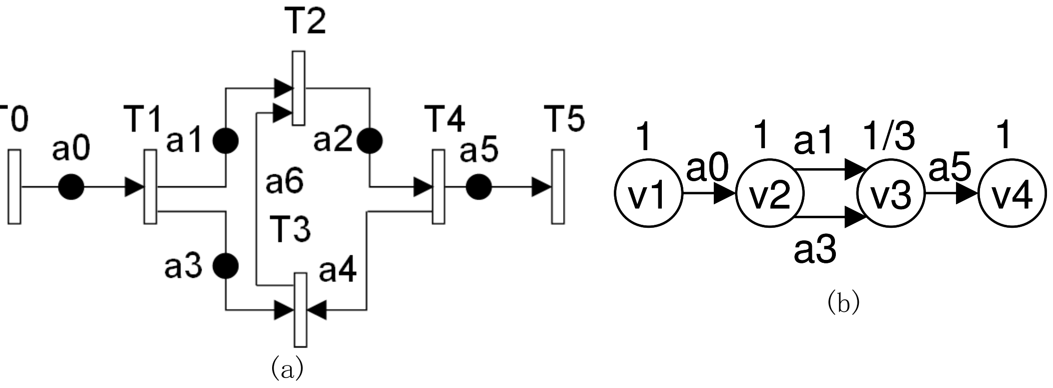


Figure 6.3: (a)A PBPG model and (b) its SCCG

Theorem 13. *SCCG is a weakly connected Directed Acyclic Graph (DAG).*

This follows directly from the definition. If there is a cycle in SCCG such as $v0 \rightarrow v1 \rightarrow v2 \rightarrow v0$, then $\{v0, v1, v2\}$ will be merged into one SCC.

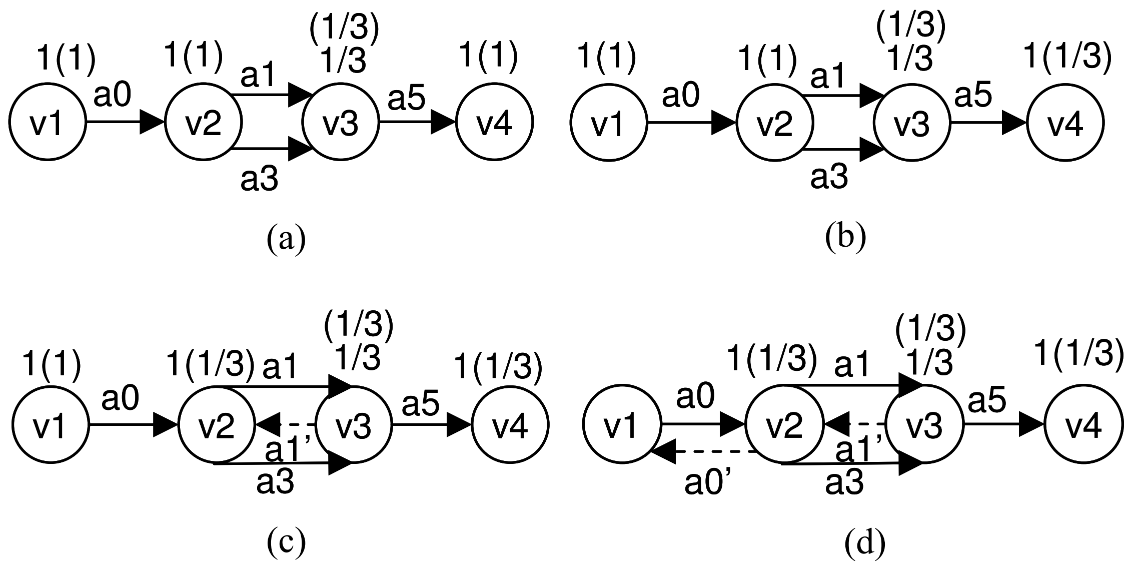


Figure 6.4: A stabilization process

In Fig. 6.2, $MinV = \{v3\}$ and $\delta(v3) = 1/3$. Back pressure arcs are required for these communication arcs where the data flows are from faster SCCs (with higher δ) to slower SCCs (with lower δ as indicated by Theorem 12. When the proper back pressure arcs are added and the throughputs are balanced, the system becomes bounded under ASAP scheduling. Then all SCCs will have the same throughput δ_0. We refer this process of making a system bounded as the *stabilization* process.

Definition 14 (Stabilized SCCG). *For $\forall v \in V_{scc}$, v is stabilized iff $\delta'(v) = \delta_0$. An SCCG is stabilized iff $\forall v \in V_{scc}$, v is stabilized.*

The stabilization process for the SCCG in Fig. 6.2 and the throughput δ''s change are shown in Fig. 6.2. $\delta'(vi)$ are shown in parentheses.

1. Initially in Step(a), $\delta'(vi) = \delta(vi)$ and $MinV = \{v3\}$.
2. In Step(b), since data flow is from $v3$ to $v4$, $\delta'(v4)$ is slowed down to $\delta'(v3) = 1/3$ without back pressure arc for $a5$. $v4$ is stabilized.
3. In Step(c), since $\delta'(v2) > \delta'(v3)$ and data flow is from $v2$ to $v3$, according to Theorem 12, back pressure arc $a1'$ must be added to $a1$. This merges $v2$ and $v3$ into one SCC so that $\delta'(v2) = \delta'(v3)$. Thus, $v2$ is stabilized.
4. Similarly in Step(d), $v1$ is stabilized by adding back pressure arc $a0'$ to $a0$ and the SCCG is stabilized.

It can be observed that as a result of stabilization, each node in this SCCG is *reachable* (see Definition 15) from $MinV$ through a path which consists of originally exiting forward arcs or added back pressure arcs. In the following context, we are going to show that under Assumption 1, this is actually a necessary and sufficient condition for the stabilization of an SCCG. The formal definition of reachability is given below:

Definition 15 (Reachability)**.** *Let v be a node in V, and let $V1$ be a subset of V. Node v is* reachable *from $V1$ iff either $v \in V1$, or $\exists a_0, a_1, \ldots, a_k$ such that $\exists u \in V1$ and $u \xrightarrow{a_0} u_1 \xrightarrow{a_1} u_2 \longrightarrow \ldots, \overset{a_k}{\to} v$ (i.e., there is a directed path from u to v). Here, a_i $(i = 0, 1, \ldots, k)$ is either a forward or an added back pressure arc.*

During stabilization process, added back pressure arcs might merge some SCCs into one bigger SCC. In the end, a different SCCG$_f(V_{sccf}, A_{sccf}, \delta_f, \delta'_f, MinV_f)$ is formed compared to the original SCCG$(V_{scc}, A_{scc}, \delta, \delta', MinV)$.

Lemma 1. *In SCCG$_f(V_{sccf}, A_{sccf}, \delta_f, \delta'_f, MinV_f)$, $\forall a \in A_{sccf}$, a is a forward arc, i.e., $L(a) = F$.*

Proof: Suppose $\exists a'(v1, v2) \in A_{sccf}$ such that $L(a) = B$, where $v1$,$v2$ are two different SCCs in V_{sccf}. Since back pressure arc is always added along a existing forward arc, there must be a $a(v2, v1) \in A_{sccf}$. Therefore, $v1$ and $v2$ can be merged into one SCC. □

Theorem 14. *SCCG$_f(V_{sccf}, A_{sccf}, \delta_f, \delta'_f, MinV_f)$ is stabilized iff $\forall v \in V_{sccf}$, v is reachable from $MinV_f$.*

Proof: "$\Leftarrow$": If $v \in MinV_f$, v is stabilized. Otherwise, $\exists u \in MinV_f$ such that there is a directed path from u to v denoted as $u \xrightarrow{a_0} u_1 \xrightarrow{a_1} u_2 \longrightarrow \ldots, \overset{a_k}{\to} v$. By Lemma /reflemma1, $L(a_0) = F$, then u_1 is slowed down which makes $\delta'_f(u1) = \delta_f(u)$. Similarly, $L(a_1) = F$ which makes $\delta'_f(u_2) = \delta'_f(u_1)$. Continue this procedure along this path until v is reached and $\delta'_f(v) = \delta_f(u)$. By Definition 6.2, SCCG$_f$ is stabilized.
"$\Rightarrow$": Suppose SCCG$_f$ is stabilized, then $\forall$ $v \in V$ is stabilized. Let $\delta'(v) = \min_{v_i \in V_{sccf}}\{\delta(v_i)\}$. If initially, v is in $MinV_f$, the proof is done. Otherwise, we can assume that v has no predecessor in SCCG$_f$. If not, there must be an arc $a_1(u_1, v) \in A_{sccf}$, where $u_1 \in V_{sccf}$. Since SCCG$_f$ is a DAG, we can always trace back to a node $u \in V_{sccf}$ with no predecessor. If $u \in MinV_f$, the proof is done since v is reachable from u. If not, there is no node which can slow down u and u is not stabilized which contradicts fact the SCCG$_f$ is stabilized. □

When back pressure arcs are gradually added, this theorem can be used to determine the boundedness of the current modified SCCG. However, it can not be directly utilized for back pressure optimization, since the topology of the modified SCCG in the process depends on the selection of added back pressure. It will be more convenient if boundedness can be analyzed by checking reachability in the original SCCG which does not change during adding back pressures.

Recall that in the modified graph SCCG$_f$, each node $v \in V_{sccf}$ is a strongly connected graph merged from one or more strongly connected sub-graph represented by nodes $u \in V_{scc}$. We have a relation between SCCG$_f$ and SCCG such that:

Theorem 15. *Let SCCG$_f(V_{sccf}, A_{sccf}, \delta_f, \delta'_f, MinV_f)$ be a SCCG obtained from SCCG$(V_{scc}, A_{scc}, \delta, \delta', MinV)$ by adding back pressure arcs. With Assumption 1, for each $v_f \in MinV_f$, $\exists v \in MinV$ such that v is a subgraph of v_f.*

Proof: With Assumption 1, we can directly get $\delta(MinV_f) = \delta(MinV)$. Suppose $\exists v_{f0} \in MinV_f$, v_{f0} doesn't include any subgraphs which belong to $MinV$. By definition of $MinV_f$, $\delta(v_{f0}) = \delta(MinV_f) = \delta(MinV)$. On the other hand, suppose v_{f0} is merged from $u_0, u_1, \ldots, u_k \in V_{scc}$ through back pressure arcs. Since $u_0, u_1, \ldots, u_k$ are not in $MinV$, $\delta(u_k) > \delta(MinV)$. Then according to Assumption 1, $\delta(v_{f0}) = \min \delta(u_k) > \delta(MinV)$, $k = 0, 1, \ldots, m$. It contradicts with $\delta(v_{f0}) = \delta(MinV)$. □

Now, with Assumption 1, we are able to prove that:

Theorem 16. *Given a PBPG G and its SCCG$(V_{scc}, A_{scc}, \delta, \delta', MinV)$, added back pressure arcs transfer SCCG to graph SCCG1$(V_{scc}, A1_{scc}, \delta, \delta', MinV)$. SCCG1 is the graph with newly added back pressure arcs in $A1_{scc}$ without merging new SCCs in SCCG. With Assumption 1, SCCG1 is bounded iff $\forall v \in V_{scc}$, v is reachable from $MinV$.*

If we merge these SCCs in SCCG1 and recalculate the each SCC's throughput, we will get SCCG$_f(V_{sccf}, A_{sccf}, \delta_f, \delta'_f, MinV_f)$ as mentioned before. Since we have already proved Theorem 14, we only need to prove the alternative Theorem 17.

Theorem 17. *Given a SCCG1$(V_{scc}, A1_{scc}, \delta, \delta', MinV)$ and SCCG$_f(V_{sccf}, A_{sccf}, \delta_f, \delta'_f, MinV_f)$ as mentioned above, let $u \in V_{scc}$ and u is merged into a $v \in V_{sccf}$ in SCCG$_f$. With Assumption 1, u is reachable from $MinV$ in SCCG1, iff v is reachable from $MinV_f$ in SCCG$_f$.*

Proof: "$\Leftarrow$": If $v \in MinV_f$, according to Theorem 15, in SCCG1, $\exists u_0 \in MinV$ such that u_0 is a sub-SCC of v. Since u is also a sub-SCC of v, u and u_0 are either the same or u is reachable by u_0. The proof is done. If v is not in $MinV_f$, $\exists v_0 \in MinV_f$ and v is reachable from v_0. Similarly, v_0 contains a sub-SCC in $u_0 \in MinV$. By the definition of SCC, u is also reachable from u_0.

"$\Rightarrow$": If $u \in MinV$, $\delta(u) = \delta(MinV)$. According to Assumption 1, $\delta(MinV_f) = \delta(MinV) = \delta(u) \geq \delta(v)$. By the definition of $MinV_f$, $\delta(v) = \delta(MinV_f)$ which indicates that $v \in MinV_f$, the proof is done. If u is not in $MinV$, $\exists u_0 \in MinV$ such that u is reachable from u_0 in SCCG1. Let u_0 is merged into a $v_0 \in V_{sccf}$, for the same reason as the previous case, we can derive that $v_0 \in MinV_f$. By the definition of SCC, v is also reachable from v_0. The proof is done. □

Theorem 16 indicates that one can analyze a PBPG's boundedness by checking reachability in its original SCCG with respect to $MinV$, which is more convenient for back pressure optimization.

6.3 Algorithm for minimizing back pressure arcs (MBPA)

Most SCCGs do not satisfy the requirements of Theorem 16. Not all nodes in V_{scc} are reachable from $MinV$. For example in Fig. 6.5(a), $MinV = \{v4, v6\}$, only node $v8$ is reachable from $MinV$ and this system is not stable. To make it stable, back pressure arcs should be added to make each node $v \in V_{scc}$ reachable from $MinV$. As discussed in Section 1.1, inserting back pressure edges results in area overhead for the control logic. Therefore, it is important to minimize the number of back pressure edges added to stabilize the system. This leads to a graph theoretic optimization problem. It seems that adding arcs other than back pressure arcs can also stabilize the system. For example, in Fig. 6.5(a), simply adding two arcs $v8 \rightarrow v0$ and $v8 \rightarrow v9$ shown as dashed arrows could make $\forall v \in V_{scc}$ reachable from $MinV$. However, in a real system, the wires represented by these arcs are not related to the existing LIS's data path and may have large latencies.

For this reason, we make a pessimistic assumption by only considering the addition of back pressure arcs along existing forward arcs. The minimization of back pressure arcs (MBPA) problem can be formulated as the graph problem shown below:

Minimization of Back Pressure Arcs (MBPA)

GIVEN: A weakly connected acyclic SCCG$(V_{scc}, A_{scc}, \delta, \delta', MinV)$.

REQUIREMENT: Extend A_{scc} to A'_{scc} (with restriction that $\forall a' \in A'_{scc} \backslash A_{scc}$, $v \xrightarrow{a'} u$, it must be that $\exists a \in A_{scc}$, $u \overset{a}{\rightarrow} v$) such that $\forall v \in V_{scc}$, v is reachable from $MinV$ in the extended graph SCCG'$(V_{scc}, A'_{scc}, \delta, \delta', MinV)$.

OBJECTIVE: Minimize $|A'_{scc}| - |A_{scc}|$.

The problem specification allows only back pressure arcs to be added. The following lemma points out a simplification to the problem.

Lemma 2. *In the statement of MBPA, one can assume without loss of generality that $MinV$ contains exactly one node.*

Proof: Suppose $MinV$ contains two or more nodes. Construct a new directed graph SCCG1$(V1_{scc}, A1_{scc}, \delta, \delta', MinV)$ from SCCG in the following manner.
(a) Let $V1_{scc} = V_{scc} \cup \{vx\}$, where vx is a new node.
(b) Let $A1_{scc} = A_{scc} \cup \{(vx, v_1) \ : \ v_1 \in MinV\}$.
This construction introduces a new node vx and adds a directed edge from vx to each node in $MinV$. Let $MinV1$ of SCCG1 have only one node, namely vx. SCCG1 is also weakly connected and acyclic. Further, if a node v is reachable from some node in $MinV$ in SCCG, then in SCCG1,

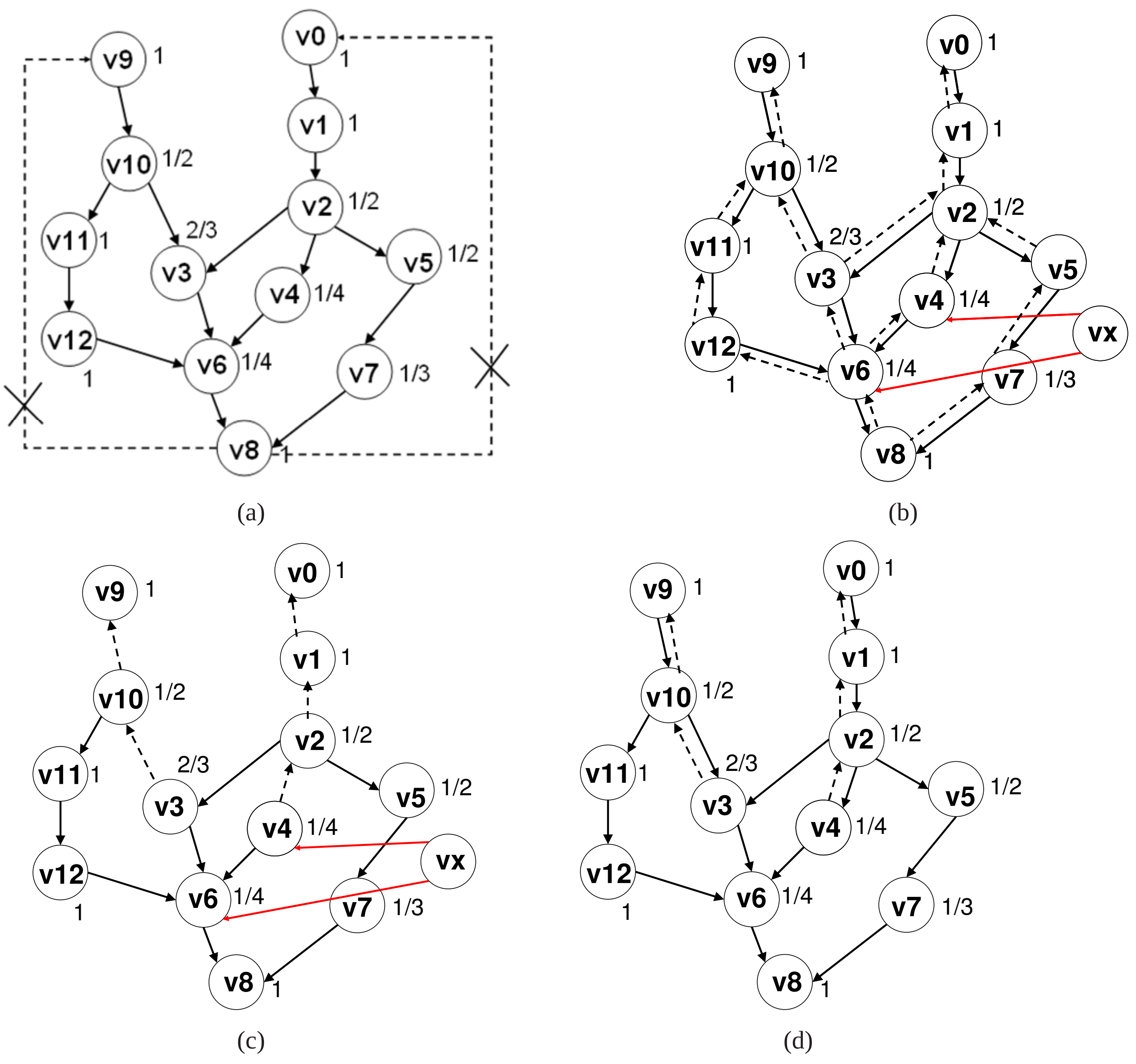


Figure 6.5: (a) A given SCCG, (b) extended SCCG1, (c) T: MCA of SCCG1 with root vx, (d) optimal extension SCCG$_o$ obtained from MBPA

v is reachable from vx. Thus, any optimal extension of SCCG making $\forall v \in V_{scc}$ reachable from $MinV$ is an optimal extension of SCCG1 making $\forall v \in V1_{scc}$ reachable from $MinV1 = \{vx\}$ and vice versa. □

In view of the above lemma, we can focus on the version of the MBPA where there is just one node in $MinV$. We will now show that the MBPA problem can be transformed into the classic *Minimum Cost Arborescence* (MCA) problem for directed graphs [75]. A formal definition of the concept of *arborescence* is as follows.

Definition 16. *Let $G(V, A)$ be a directed graph and let r be a node in V. An* ***arborescence*** *of G with respect to r is a subgraph $T(V, F)$ such that the following conditions hold.*

1. *$F \subseteq A$.*
2. *T is a spanning tree for G if the directions on the edges in F are omitted.*
3. *For every node $v \in V$, there is a directed path in T from r to v.*

If T is an arborescence of a graph G with respect to a node r, one can draw T with r as the root. For this reason, we will use the phrase "arborescence rooted at r" in preference to "arborescence with respect to r". Given a directed graph $G(V, A)$ and a node r, the following lemma from [48] gives a necessary and sufficient condition for G to have an arborescence rooted at r.

Lemma 3. *Given a directed graph $G(V, A)$ and a node r, G contains an arborescence rooted at r if and only if for every node $v \in V$, there is a directed path from r to v in G.* □

Suppose $G(V, A)$ is a directed graph with a nonnegative cost w_e for each directed edge $e \in A$. If $T(V, F)$ is an arborescence of G rooted at a node r, then the cost of T is the sum of the costs of the directed edges in F. We can now define the minimum cost arborescence (MCA) problem.

Minimum Cost Arborescence (MCA)
GIVEN: A directed graph $G(V, A)$ with nonnegative edge costs and a node $r \in V$.
REQUIREMENT: Find a minimum cost arborescence of G rooted at r, if such an arborescence exists.

The MBPA problem can now be solved using the algorithm shown in Fig. 6.6. We can now prove the correctness of the algorithm.

Theorem 18. *The solution provided by the algorithm shown in Fig. 6.6 is an optimal solution for MBPA problem.*

Proof: To prove the theorem, we need to show the following:

1. There is an arborescence $T(V_{scc}, F)$ rooted at vx for the graph SCCG1 that results at the end of Step 1) of the algorithm.

Algorithm

Input: A weakly connected directed graph SCCG$(V_{scc}, A_{scc}, \delta, \delta', MinV)$.

Output: An optimal extension SCCG$_o(V_{scc}, A_o, \delta, \delta', MinV)$ of SCCG.

Steps:

1. Construct an edge weighted directed graph SCCG1$(V1, A1, \delta, \delta', MinV)$ from SCCG$(V_{scc}, A_{scc}, \delta, \delta', MinV)$ as follows.
   (a) For each edge $(u, v) \in A_{scc}$, add both the directed edges (u, v) and (v, u) to $A1$.
   (b) The costs of edges in $A1$ are determined as follows: if $(u, v) \in A1 \backslash A_{scc}$, $w(u, v) = 1$, otherwise $w(u, v) = 0$.
   (c) Add a node vx to V_{scc} to get $V1$ so that $V1 = V_{scc} \cup \{vx\}$ and edges (vx, u) to $A1$, if $u \in MinV$. Let $w(vx, u) = 0$.
2. Obtain a minimum cost arborescence $T(V_{scc}, F)$ of SCCG1$(V1, A1, \delta, \delta')$ with root vx.
3. Remove edges (u, v) with $w(u, v) = 1$ from $A1$, if $(u, v) \notin F$.
4. Remove node vx from $V1$ to get back to V_{scc}, and remove edges (vx, u) from $A1$ for $\forall u \in MinV$ to get A_o; the resulting graph is SCCG$_o(V, A_o, \delta, \delta', MinV)$.

Figure 6.6: Steps of the Algorithm for MBPA

2. Graph SCCG$_o(V_{scc}, A_o, \delta, \delta', MinV)$ satisfies the requirements of the MBPA problem.
3. The number of edges in $|A_o| - |A_{scc}|$ is a minimum.

We prove these parts separately.

Part 1: From Lemma 3, it is only required to show that in SCCG1, there is a directed path from vx to each node in $V1 - \{vx\} = V_{scc}$. Since SCCG is weakly connected, there is an undirected path from each $v0 \in MinV$ to each node in V_{scc} in the graph obtained by erasing the edge directions in SCCG. For each edge $\{u, v\}$ of this undirected path, the graph SCCG1 contains both the directed edges (u, v) and (v, u) by Step 1a). Therefore, in SCCG1, there is a directed path from $v0$ to each node in V_{scc}. On the other hand, by Step 1c), a directed path from vx to $v0$ is created. Thus, there is a directed path from vx to each node in V_{scc}.

Part 2: We argue that the edge set A_o of SCCG$_o$ satisfies the requirements of MBPA problem (see Definition of MBPA). By Step 3), A_o contains all the edges in A_{scc} plus all the edges of cost 1 from the arborescence T. Further, for each edge (v, u) of cost 1 in T, the reverse edge (u, v) is in A_{scc}. Thus, $A_{scc} \subseteq A_o$ and for each edge $(u, v) \in A_o$, either (u, v) or (v, u) is in A_{scc}. Further, in T, there is a directed path from vx to each node in $V_{scc} = V1 - \{vx\}$. By Lemma 2, V_{scc} is reachable from $MinV$. Therefore, SCCG$_o$ satisfies the requirements of MBPA problem.

Part 3: Here, we need to show that the number of new edges added to A_{scc}, that is, $|A_o| - |A_{scc}|$ is a minimum. By Lemma 2, we only need to consider the case when $MinV$ has just one node. Thus, we can assume $MinV = \{vx\}$ in the following argument. Let OPT$_{\text{aug}}$ denote the minimum number of edges to be added to A_{scc} to obtain an optimal solution of MBPA. Let OPT$_{\text{arb}}$ denote the cost of the optimal arborescence T produced in Step 2) of the algorithm. Note that the only edges added to A_{scc} to produce A_o are the edges of cost 1 in T by Step 3). Thus, OPT$_{\text{arb}}$ = $|A_o| - |A_{scc}|$. Thus, we can complete the proof of Part 3 by showing that OPT$_{\text{aug}}$ $\geq$ OPT$_{\text{arb}}$.

We prove this by contradiction. Suppose OPT$_{\text{aug}}$ $<$ OPT$_{\text{arb}}$. Let SCCG$^*(V_{scc}, A^*, \delta, \delta', MinV)$ be an optimal solution of SCCG. Since SCCG* is optimal, OPT$_{\text{aug}}$ = $|A^*| - |A_{scc}|$. Also, by the requirement of MBPA, in SCCG*, there is a directed path from vx to each node in V_{scc}. Therefore, by Lemma 3, SCCG* contains an arborescence T^* rooted at vx. Note that T^* contains at most OPT$_{\text{aug}}$ directed edges which are not in A_{scc}. Further, each edge in A^* is also in $A1$. Therefore, T^* is also an arborescence rooted at vx for SCCG1. Further, because of our choice of edge costs in SCCG1, the cost of T^* is at most OPT$_{\text{aug}}$. Since we assumed that OPT$_{\text{aug}}$ $<$ OPT$_{\text{arb}}$, we reach the conclusion that there is an arborescence T^* for SCCG1 whose cost is *less than* OPT$_{\text{arb}}$. This contradicts the optimality of the arborescence T constructed in Step 2). This completes the proof of Part 3 and also that of the theorem. □

Fig. 6.5(d) shows the back pressure arcs added by our algorithm with dashed arrows. The algorithm adds 5 back pressure edges. In contrast, 14 back pressure edges get added if one inserts a back pressure edge for every $a \in A_{scc}$.

6.4 Experimental results

The complexity of MBPA includes the complexity of searching for all SCCs, computing each SCC's throughput and solving MCA problem. SCCs of a PBPG (T, A, M_0, B) can be found in time $O(|T| + |A|)$ using Tarjan's algorithm [74]. Suppose each scc_k has n_k transitions and e_k arcs ($\sum n_k = |T|$ and $\sum e_k < |A|$), the throughput for scc_k can be calculated as the minimum cycle mean by the algorithm given in [34] with the time complexity $O(n_k(n_k + e_k))$ for the worst case. Therefore the complexity of constructing the SCCG$(V_{scc}, A_{scc}, \delta, \delta', MinV)$ from the PBPG has a complexity of $O(\sum n_k(n_k + e_k)) + O(\sum n_k + |A|$ which is $O(\sum n_k(n_k + e_k) + |A|)$. On the other hand, MCA problem has polynomial time algorithms [37, 27, 10]. We choose an open source package from [1], which has time complexity $O(|A_{scc}| \log |V_{scc}|)$ for sparse graphs and $O(|V_{scc}|^2)$ for normal graphs. Therefore, the MBPA has a worst case complexity of $O(\sum n_k(n_k + e_k) + |A| + |V_{scc}|^2)$. Three groups of benchmarks are used in this experiment. $(s27 \sim s5378)$ are ISCAS'89 and $(b04 \sim b13)$ are ITC benchmarks. We model each gate as a transition in PBPG model. Even though LISs are not generally synthesized at such a fine level of granularity, these benchmarks can still provide complex graph topology to evaluate our methods. Interconnect latency between two transitions is randomly generated, and relay stations are inserted based on the latency information. We implement the technique MBPA on our tool LOLA-LIS, an extension of LOLA (alow level petri-net analyzer [67]) for LIS design. Experiments are done on a PC with Intel CPU T1300 at 1.66GHz with 1G memory.

In Table 6.1, column $|A_B|$ shows the number of required back pressure arcs computed by MBPA. Column $|A_B|/|A|$ represents the ratio between the number of minimal back pressure arcs and the number of overall inter-SCC arcs. This ratio is less than 27.3% for all benchmarks and is below 10% when the SCCG becomes large. The result demonstrates that only a small percentage of inter-SCC arcs need back pressure.

Table 6.1: Experiment results of MBPA

Bench	LIS Info.		SCC Info.				Throughput Info.			Performance	
marks	$\|T\|$	$\|A\|$	$\|V_{scc}\|$	$\|A_{scc}\|$	$\|A_B\|$	$\|A_B\|/\|A\|$	δ_{HS}	δ_{MBPA}	δ_U	Exec.(s)	Mem.(M)
s27	18	22	9	16	6	27.3%	0.83	0.83	0.83	0.06	2.95
s298	161	283	89	187	10	3.5%	0.75	0.75	0.75	0.09	3.11
s344	216	316	66	119	11	3.5%	0.68	0.68	0.68	0.08	3.02
s349	217	320	65	120	13	4.1%	0.74	0.74	0.79	0.08	3.00
s382	188	333	104	198	10	3.0%	0.71	0.78	0.80	0.13	3.76
s386	205	386	91	204	12	3.1%	0.67	0.78	0.80	0.11	3.21
s400	220	375	105	203	10	2.7%	0.67	0.71	0.77	0.13	3.16
s444	211	379	98	197	11	2.9%	0.67	0.73	0.80	0.29	3.77
s641	506	631	240	382	67	10.6%	0.55	0.55	0.55	0.31	4.53
s713	523	686	240	383	66	9.6%	0.63	0.73	0.75	0.33	5.21
s820	396	846	171	484	30	3.5%	0.33	0.35	0.36	0.25	4.11
s832	397	861	168	447	30	3.5%	0.34	0.34	0.34	0.23	5.09
s1423	753	1243	205	400	52	4.2%	0.67	0.67	0.67	0.50	8.62
s1488	810	1536	442	885	13	0.8%	0.53	0.53	0.53	0.63	6.52
s1494	804	1542	439	886	13	0.8%	0.60	0.60	0.60	0.59	6.22
b04	737	1341	588	1090	13	0.6%	0.60	0.60	0.60	7.34	6.38
b05	998	1941	943	1773	1	0.02%	0.67	0.75	0.75	11.78	7.69
b06	56	98	34	60	2	2.0%	0.75	0.75	0.75	0.24	3.32
b07	441	806	117	298	2	0.4%	0.60	0.60	0.60	2.25	4.84
b08	183	331	169	244	12	2.2%	0.57	0.57	0.57	1.34	4.05
b09	170	306	59	85	2	1.0%	0.60	0.60	0.60	1.14	4.26
b10	206	376	67	121	12	5.7%	0.57	0.57	0.57	0.77	4.67
b11	770	1415	49	76	7	4.5%	0.50	0.50	0.50	3.48	27.10
b12	1076	2094	144	311	7	1.3%	0.64	0.64	0.64	15.55	35.72
b13	362	621	101	171	10	3.3%	0.67	0.67	0.67	1.73	6.07

We must point out that adding back pressure edges may affect the system's throughput by creating new critical cycles as observed in [31, 57] if the buffer sizes of inter-SCC arcs are not properly chosen. It may not satisfy Assumption 1 because of newly formed critical cycles. For example in Fig. 6.7, $p0'$ is a back pressure arc added to slow down SCC$_{t5\rightarrow t6\rightarrow t7}$ and bring down its throughput from $2/3$ to $3/5$. However, a new critical cycle $t1 \rightarrow t7 \rightarrow t5 \rightarrow t8 \rightarrow t3 \rightarrow t2 \rightarrow t1$ is formed through arc $p0'$, which brings down the system's throughput to $1/2$ instead of the expected $3/5$. This problem also exists in hand-shake based protocols where all arcs have back pressure. However, we now show that the throughput of the LIS synthesized by MBPA is not less than the one produced by a hand shaking based protocol.

Theorem 19. *Given an LIS P, let P_{MBPA} denote the result of synthesizing the system P by MBPA and let P_{HS} denote the system synthesized using hand shaking for every link of P. The throughput of P_{MBPA} is at least as large as the throughput of P_{HS}; symbolically, $\delta(P_{MBPA}) \geq \delta(P_{HS})$.*

Proof: Let A_{MBPA} be the set of arcs in P_{MBPA} and A_{HS} be the set of arcs in P_{HS}. By definition, $A_{MBPA} \subseteq A_{HS}$. If the throughput of P_{MBPA} is determined by its critical cycle C_0 which is formed by arcs $a_0 \rightarrow a_1 \ldots \rightarrow a_m \rightarrow a_0$, where $a_0, \ldots, a_m \in A_{MBPA}$, then $a_0, \ldots, a_m$ also belong to A_{HS} and C_0 is also a cycle of P_{HS}. On the other hand, as the number of initial tokens in C_0 depends only on P (number of IP blocks), $\delta(C_0)$ is same in both P_{MBPA} and P_{HS}. By definition of throughput, $\delta(P_{HS}) \leq \delta(C_0) = \delta(P_{MBPA})$. □

For now, we compare the throughputs before and after using back pressure arcs. Let δ_U be the throughput determined by $\text{Min}_{v_i \in V}\{\delta(v_i)\}$ which is the maximum throughput the system can achieve. Let δ_{MBPA} be the throughput computed by ASAP scheduling according to Equation (2.5) after back pressure arcs are added based on the solution to the MBPA problem. In Table 6.1, δ_{HS} is the throughput when back pressure arcs are added for all $a \in A$. For most of our benchmarks, these three results are same. In "s349", "s382", "s386", "s400", "s444", "s713" and "s820", δ_{MBPA} decreased a little from δ_U but is still better than or equivalent to δ_{HS}, which coincides with the claim of Theorem 19 that it performs better than adding back pressure arcs to every $a \in A$.

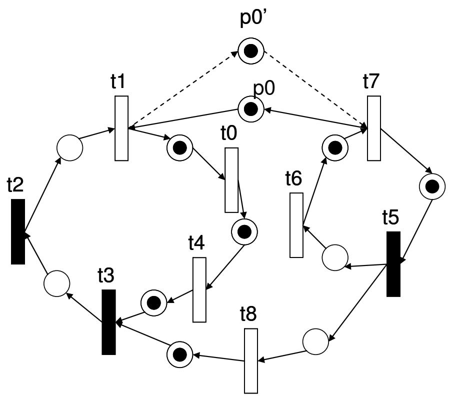


Figure 6.7: An example where the throughput is deteriorated

6.5 Conclusions

Observing that back pressure must be used to avoid overflow when a faster SCC sends data to a slower SCC, we extended the analysis to general LIS by proposing SCCG. We have formally shown that preventing overflow of an LIS requires that each node v in SCCG be reachable from $MinV$ by adding back pressure arcs. The problem of finding a minimal set of back pressure arcs was formalized as the Minimization of Back Pressure Arcs (MBPA) problem, which can be solved efficiently by a transformation to the Minimum Cost Arborescence (MCA) problem. Experimental results demonstrate that our algorithm is efficient and the number of back pressure arcs added is only a small percentage of the overall inter-SCC arcs. Further, the throughput of our method is no worse than that of a realization using a hand-shake based protocol.

Chapter 7

Implementation of the protocol

In this chapter, we present an implementation of LIS which refines its PBPG model. If we consider a state of PBPG as a $|T|$-bit vector $S(s_0, s_1, \ldots, s_{|T|-1})$ such that bit $s_k = 1$ (s_k=0) indicates transition t_k is enabled (disabled) in the given instant. MBPA, along with throughput improvement techniques such as RMILP and LMILP, guarantees the boundness of the PBPG and optimize its throughput. As a result, this PBPG will have a periodic state sequence as $S = S_u(Sv)^\omega$, where S_u is the initial sequence and S_v is the iterative part when ASAP scheduling is used. This is due to the fact that a PBPG is deterministic and has finite reachable states if it is bounded (see the discussion of Theorem 1). Accordingly, each transition will also have a periodic firing sequence after an initial phase. It behaves as if it was clock gated by a periodic sequence. However, if the clock gating sequences were not implemented properly, it could be problematic because of the following issues.

- Generating the clock gating sequences for every block from a centralized module will cause routing problems.
- Storing the clock gating sequences in local shift registers may lead to more area overhead than it saves, especially when the period of the clock gating sequence is long.

By ASAP scheduling, each block is executed as long as all of its inputs are available. (Firing constrain by one arc's output buffer size can also be regarded as checking the input availability of its complementary arc (back pressure arc). See the discussion about the strict firing rule and weak firing rule in Section 2.1). Intuitively, we only need to transfer a signal which represents data validity between different blocks to clock gate that block. It simulates the tokens's passing from one transition to another. A simple example shown in Fig. 7.1 gives a hint on such implementation. Two IP blocks IP1 and IP2 are connected through a relay station RS and are clock gated by "valid" signals v1, v2, v3 stored in a flip-flop. The clock gating logic is not explicitly shown for the sake of simplicity. (It is similar to the one used in Fig. 7.2(a).) Initially, v1 = 1, v2 = 0 and v3 = 1; thus, in the first clock cycle RS and IP2 are enabled while IP1 is not (see the left part of Fig. 7.1). In

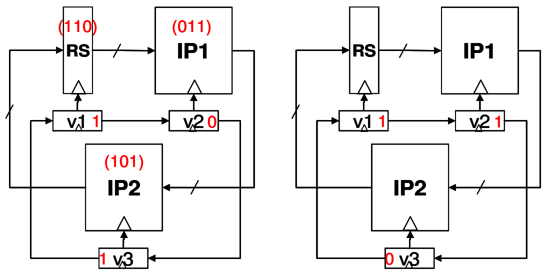


Figure 7.1: Implementation of ASAP scheduling for LIS

the second clock cycle, the valid bits are transferred to the next stage such that v1 = 1, v2 = 1 and v3=0. Accordingly, RS and IP1 are enabled but IP2 is not (see the right part of Fig. 7.1). Thus, over time, RS, IP1 and IP2 will have periodic schedulings (110), (011) and (101) respectively. This simple example shows that the periodic scheduling can be formed by storing and transferring only the information based on input validity. The scheduling sequence of one block is one bit right rotation of its previous block's scheduling sequence. In this example, all three arcs belong to A_U in its PBPG model and their buffer sizes are "1". They can be implemented as a register clock gated by a valid bit, which is also stored in another flip-flop. However, for other arcs in PBPG, their implementations depends on their type and sizes which are not so obvious.

Firstly, we need to determine buffer size $B(a), a \in A$, which can be obtained with respect to three cases:

1. $a \in A_U$ and a is not an incoming arc for a *join transitions* which is a transition with multiple incoming arcs. No more than one token can reside on a at any time according to ASAP scheduling, so $B(a) = 1$;

2. $a \in A_F$ which has back pressure. $B(a)$ is determined by solving the RMILP or LMILP formulation discussed in Section 8.3 and Section 8.4.

3. $a \in A_U$ and a is an incoming arc for a join transition. More than one token may accumulate on a due to barrier synchronization. However, its maximum token number is bounded if the PBPG is bounded. $B(a)$ can be derived from the firing sequence of v_i, v_j ($a(v_i, v_j)$) and its initial marking $M_0(a)$ by Equation 7.1 [78].

$$B(a) = \max_{k>0} |s_i[1...k]|_1 - |s_j[1...k]|_1 + M_0(a) \tag{7.1}$$

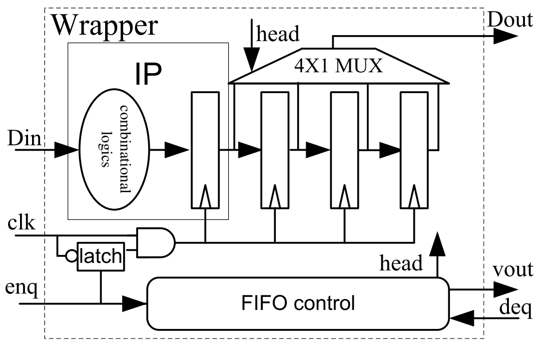
where s_i and s_j are the scheduling sequences of t_i and t_j. $|s_i[1...k]|_1$ represents the number of "1"s included in the k-length prefix of s_i. Since s_i and s_j are periodic, $B(a)$ can be computed by enumerating k in $[0, |v| + |u| - 1]$, where $|v|$ and $|u|$ are the length of initial sequence and periodic sequence of the schedulings. $B(a)$ is the maximal number of tokens that are produced by t_i, yet not consumed by t_j, representing the upper bound on tokens that will reside on arc a at any time.

Forward arc a models the buffer of ${}^\bullet a$ so the implementation of a is considered as part of the wrapper for ${}^\bullet a$. a should have unit latency irrespective of $B(a)$ according to [15] to preserve performance. The implementation of a can be classified into three cases: (1) $a \in A_U$ and $B(a) = 1$; (2) $a \in A_U$ and $B(a) > 1$; (3) $a \in A_F$. Case (1) can be implemented as the same way as shown in Fig. 7.1. Case (2) and case (3) can be implemented as bypassable synchronous first in first out (FIFO) queues. We use FIFO(n) and FIFO$_B(n)$ to denote a's implementation of case (2) and case (3) if $B(a) = n$. In fact, case (1) can be regarded as FIFO(1).

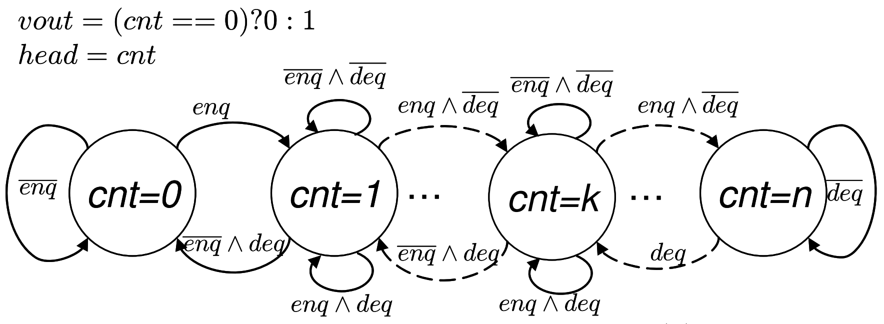
Here we focus on the implementation of FIFO$_B(n)$ and FIFO(n) is a simplified version of FIFO$_B(n)$ without detecting the "full" condition.

7.1 Implementation of FIFO(n)

Fig. 7.2(a) shows the schematic of FIFO(4). It consists of a four stage selective shift register and a *FIFO controller* (shown in Fig. 7.2(a) and Fig. 7.2(b)). The functionality of the FIFO controller for fifo4 is described by an FSM shown in Fig. 7.2(b). It guarantees that the outgoing block can read the first valid data in queue. It is unnecessary for this data to travel through all stages of the queue, if the following stages are empty or contain invalid data. Otherwise, the throughput of system is decreased due to constant delay. In particular, control of FIFO(4) has a counter cnt, indicating the number of valid data in queue. It receives enq (deq) from neighboring stages, updates $head$ and generates "$vout$". $enq = 1$ ($deq = 1$) denotes that a valid data enters (leaves) the FIFO. enq also clock gates the shift registers. $vout$ denote whether the queue is empty ($vout = 0$). The finite state machine (FSM) of FIFO$_B(n)$'s controller is shown in Fig. 7.2(b). Computation of enq and deq depends on the interconnections with neighboring stages which will be addressed later. $cnt = i$ transits to $cnt = i+1$ ($i = 0, 1, \ldots, n-1$) when $enq\&\overline{deq}$ that a valid data enters the queue but no data is consumed. Therefore, it increases the number of data stored in the queue by 1. On the other hand, $cnt = i$ moves to $cnt = i - 1$ ($i = n, n-1, \ldots, 1$) when $\overline{enq}\&deq$ holds, which means that the outgoing block removes one data from the queue but no new data entered thus the data stored in the queue decreases by 1. For other inputs combinations, the number of valid data stored in the queue remains the same, thus cnt_i stays in cnt_i. Recall that the firing sequences (schedulings) are pre-calculated, the behavior of the synchronous queue as well as $B(a)$ are determined. Some input combinations will never appear at a particular state. For example, $enq\&\overline{deq}$ can not be true in cnt_n and $\overline{enq}\&deq$ can not be true in cnt_0, thus avoid checking "full" or "empty" on FIFO(n). cnt also updates $head$ by $head = cnt$ which selects the output via a MUX. $head$ points to the stage which stores the first valid data in queue. $vout = 1$ corresponds to $M(a) > 0$. As long as the synchronous queue is not empty ($cnt \neq 0$)



(a) The implementation of FIFO(4)


$vout = (cnt == 0)?0 : 1$
$head = cnt$
cnt=0
cnt=1
cnt=k
cnt=n
$\overline{enq}$
enq
$\overline{enq} \wedge \overline{deq}$
$enq \wedge \overline{deq}$
$\overline{enq} \wedge deq$
$enq \wedge deq$
deq
$\overline{deq}$

(b) The FSM of the FIFO controller of FIFO(n)

Figure 7.2: Schematic of FIFO(4) and the FSM of its FIFO controller

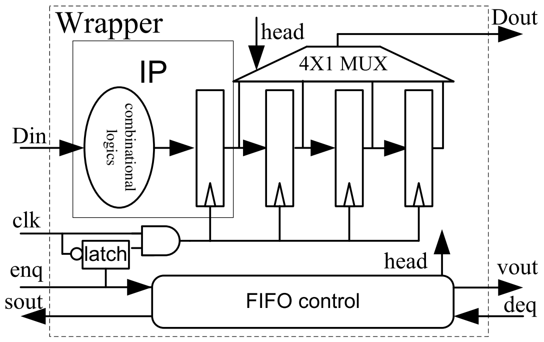
7.2 Implementation of FIFO$_B(n)$

Arc $a(t_i, t_j) \in A_F$ has back pressure is implemented as FIFO$_B(n)$ if $B(a) = n$. The FIFO controller should have a mechanism to test "full" condition and generate "stall" signals when the FIFO is full to stop the current block from execution. A schematic of FIFO(4)'s implementation is shown in Fig. 7.3(a) and Fig. 7.3(b) which is quite similar to FIFO(4). The only difference is that additional $stall$ signal is employed when $cnt = n$ denoting that the output buffer is full. $stall = 0$ relates to the situation when $M(a) < B(a)$. $stall$ is the signal carries out the back pressure information to its previous stage.

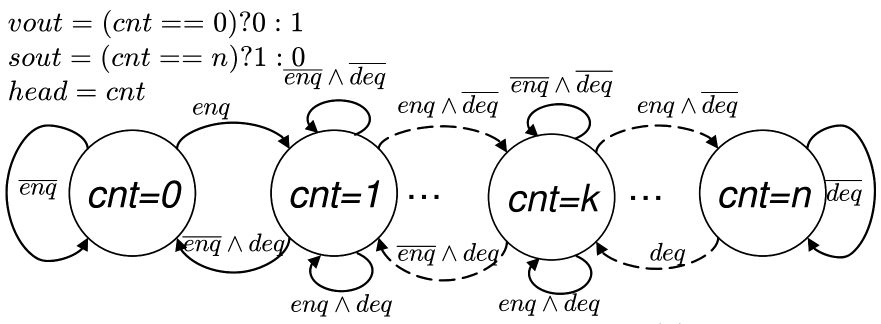
7.3 Implementation of wrappers for an IP block modeled by a join transition

Let's consider the implementation of a IP block T_{IP} with multiple incoming and outgoing arcs as $T4$ is shown in Fig. 7.4(a). The implementation follows these rules:

1. Arcs in $\bullet T_{IP}$ and $T_{IP}\bullet$ are implemented as FIFO or FIFO$_B$ as discussed previously. Given



(a) The implementation of FIFO$_B(4)$


$vout = (cnt == 0)?0 : 1$
$sout = (cnt == n)?1 : 0$
$head = cnt$
$\overline{enq}$
enq
$\overline{enq} \wedge \overline{deq}$
$enq \wedge \overline{deq}$
$\overline{enq} \wedge \overline{deq}$
$enq \wedge \overline{deq}$
cnt=0
cnt=1
cnt=k
cnt=n
$\overline{deq}$
$\overline{enq} \wedge deq$
$\overline{enq} \wedge deq$
deq
$enq \wedge deq$
$enq \wedge deq$

(b) The FSM of FIFO controller of FIFO$_B(n)$

Figure 7.3: The implementation of FIFO$_B(4)$ and the FSM of its FIFO controller

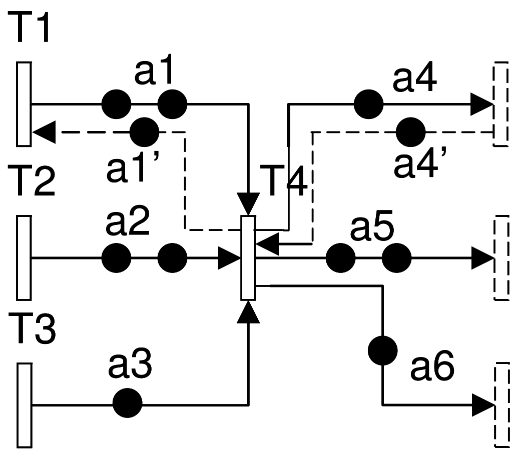
that $\bullet T4 = \{a1, a2, a3\}$, $a1 \in A_F$, $a2, a3 \in A_U$, $B(a1), B(a2) > 1$ and $B(a3) = 1$. $T4\bullet = \{a4, a5, a6\}$, $a4 \in A_F$, $a5, a6 \in A_U$, $B(a4), B(a5) > 1$ and $B(a6) = 1$. $a1$ and $a4$ are implemented as FIFO$_B(3)$ and FIFO$_B(2)$; $a2$ and $a5$ are implemented as FIFO(2); $a3$ and $a6$ are implemented as FIFO(1) as shown in Fig. 7.4(b).

2. Firing of T_{IP} removes a valid data from $a \in T_{IP}\bullet$ and produces a new data to $a \in \bullet T_{IP}$. So enq signal for a_i's ($a_i \in T_{IP}\bullet$) controller and deq for a_j's ($a_j \in \bullet T_{IP}$ and $B(a_j) > 1$) controller are related to the firing of T_{IP}.

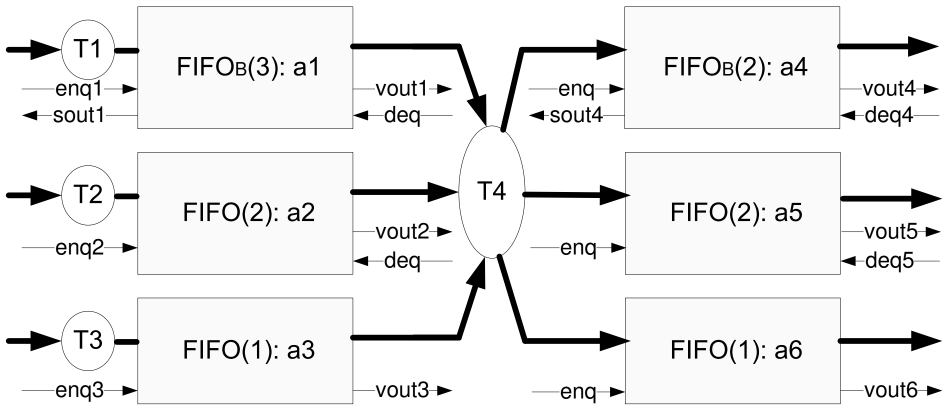
$$enq = deq = \bigwedge_{a_i \in \bullet T_{IP}} vout_i \cdot \overline{\bigvee_{a_j \in T_{IP}\bullet \cup A_F} (sout_j)} \tag{7.2}$$

For example, in Fig. 7.4(b), deq signal for $a1$ and $a2$, enq signal for $a4$, $a5$, $a6$ are generated according to:

$$enq = deq = vout_1 \wedge vout_2 \wedge vout_3 \wedge \overline{sout_4} \tag{7.3}$$



(a)



(b)

Figure 7.4: The PBPG model for a transition with multiple incoming and outgoing arcs (a); Its implementation (b)

7.4 System initialization

Finally, let's talk about the initialization. When an LIS restarts, states (cnt) of each FIFO or FIFO$_B$ should be properly reset. Initially, if arc $M_0(a) = k$ ($a \in A_F \cup A_U$), then the related FIFO (either FIFO or FIFO$_B$) should reset cnt to k, representing k valid data are stored in the FIFO. After reset, the system will behave the same way as the ASAP scheduling of its PBPG model and have a state sequence of $S = S_u(S_v)^\omega$. However, $|S_u|$ is usually much longer than $|S_v|$. Initialization from the first state of S_v rather than S_u can potentially reduce buffer sizes. The first state of S_v can be obtained by simulating the PBPG with ASAP scheduling. The comparison of buffer sizes with different initializations is discussed in Section 8.5.

Chapter 8

Throughput optimization for scheduled LIS

In Chapter 6, we have discussed the boundness problem of LIS whose graph model has multiple strongly connected components (SCC) with respect to back pressures. A technique which adds a minimal number of back pressure arcs (MBPA) to make the system bounded with ASAP scheduling was proposed. However, one remained problem is that added back pressure arcs may decrease the system's throughput if inter-SCC arcs buffer sizes are not properly chosen. In Chapter 6, we simply make them equivalent to "2" which is surely not enough to guarantee Assumption 1 as shown in Fig. 6.7. Therefore, it calls for a better technique which takes both the number of back pressure arcs and the throughput into account. This technique should get an optimal solution of back pressure arcs and also obtain a given throughput with minimum buffer sizes. There are two possible approaches to achieve this:

1. Considering the throughput constraints in MBPA: assign a weight to every forward arc a to represent the amount of reduction contributed to the final system's throughput, if a back pressure arc a' is added along a. Then modify the MBPA to find a minimum cost arborescence with respect to the weights. The solution will give a bounded system with maximal throughput. The challenge of this approach is that it is very difficult to assign such weights for every forwarding arc and represent the system's throughput as a linear combination of such weights.

2. Considering the throughput improvement after MBPA: as a result of MBPA, the system becomes an MG with both forward and back pressure arcs. Then we can increase some arcs capacity to enhance the throughput. Previous work [56] shows that one can achieve a given throughput with minimal buffer capacity increment. However, [56] is based on systems where each forward arc has a back pressure arc. We find it is direct to extend that work for the resulting system after applying MBPA. The challenge is that we need to solve a mixed integer linear programming (MILP) formulation for this approach which is NP-hard. So it may not be scalable for large systems if MILP is directly applied.

In this chapter, we take the second approach. We still use PBPG model and extend the exiting work of throughput improvement using MILP for PBPG; Meanwhile, we propose two techniques of graph abstraction to speed up solving MILP formulation and which are scalable for large systems.

8.1 Problem of throughput reduction in MBPA

We revisit MBPA and the problem left in MBPA by considering the example in Fig. 8.1 and Fig. 8.2. Fig. 8.1 shows a PBPG model for a LIS and Fig. 8.2 (a) is the related SCCG. Since the SCC $v1$ which consists of $\{T21, T22, T23\}$ has the throughput of 1/3 which is less than 4/11, the throughput of $v3$ which consists of $\{T0, T1, \ldots, T15\}$. The throughput of $v3$ is again less than than 1/2, the throughput of SCC $v5$ which consists of $\{T16, T17, T18, T19\}$. According to the algorithm of MBPA, two back pressure arcs $(T23, T8)$ and $(T2, T16)$ are added as shown in Fig. 8.1. As a result, $v1$, $v2$, $v3$, $v4$ and $v5$ are merged into one SCC. However, its throughput is $2/7$ due to the new critical cycle $T8 \rightarrow T9 \rightarrow T0 \rightarrow T1 \rightarrow T2 \rightarrow T16 \rightarrow T17 \rightarrow T18 \rightarrow T24 \rightarrow T4 \rightarrow T5 \rightarrow T6 \rightarrow T10 \rightarrow T7 \rightarrow T8$, which is less than the minimal throughput $1/3$ among $v1$, $v2$, $v3$, $v4$ and $v5$. This violates Assumption 1 and brings down the system's throughput from $1/3$ to $2/7$.

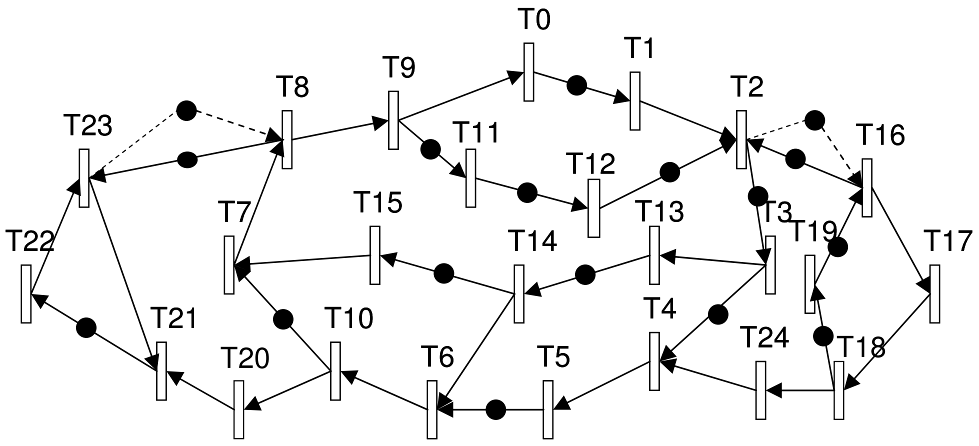


Figure 8.1: A PBPG model for a LIS G

8.2 Mixed ILP formulation (MILP) for PBPG

In [56, 15], strategies have been proposed to improve the system's throughput with minimum buffer addition. [56] was the first work which addressed this problem as *maximum performance buffer queue sizing*. Theorem 1 in [56] rewrites the constraint on system's throughput as a linear function of arcs' capacities. Based on this relation, a mixed ILP (MILP) formulation is proposed to achieve

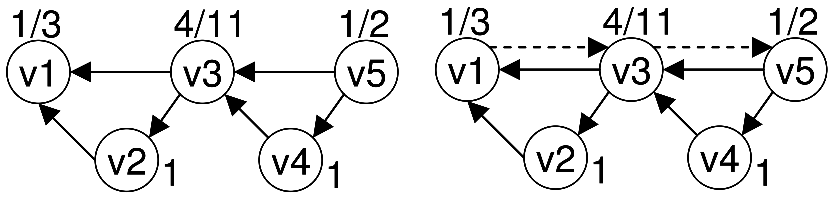


Figure 8.2: The SCCG of G (a), and its resulted SCCG' by MBPA (b)

a target throughput by increasing minimum capacities of these arcs with back pressures. Other related work modified this theorem for their respective applications. We also modify Theorem 1 in [56] and applied it to PBPG as theorem 20.

Theorem 20. *For any cycle C in a PBPG$< T, A, M_0, B >$, $\delta(C) \geq \delta^*$ iff exists a function $r : T \to \mathbb{R}$, such that $r(v_j) - r(v_i) \leq M_0(a) - \delta^*$, where $a(v_i, v_j) \in A_U \cup A_F \cup A_B$.*

The proof is also similar to the proof of Theorem 1 in [56], which is shown below.
Proof: "if:" Suppose such $r : T \to \mathbb{R}$ exists and $r(v_j) - r(v_i) \leq M_0(a) - \delta^*$ holds for any arc $a(v_i, v_j) \in A_U \cup A_F \cup A_B$. Assume C is any cycle of the graph and $C = a_0(v_0, v_1)a_1(v_1, v_2)\ldots a_N(v_N, v_0)$. Adding $r(v_j) - r(v_i) \leq M_0(a_i) - \delta^*$ along the arcs in C:

$$\begin{aligned} 0 = \textstyle\sum_{a_i \in C} r(v_i + 1) - r(v_i) \leq \sum_{a_i \in C} (M_0(a_i) - \delta^*) \\ \Leftrightarrow \delta(C) = \tfrac{\sum_{a_i \in C} M_0(a_i)}{|C|} \geq \delta^* \end{aligned} \tag{8.1}$$

"only if:" Assume for every cycle C, $\delta(C) \geq \delta^*$, which means:

$$\delta(C) = \frac{\sum_{a_i \in C} M_0(a_i)}{|C|} \geq \delta^* \Leftrightarrow \sum_{a_i \in C} (M_0(a_i) - \delta^*) \geq 0 \tag{8.2}$$

We add an additional transition s_0 to T such that $T' = T \cup \{s_0\}$ and directed arcs $a(s_0, v)$ ($\forall v \in T$) to A_U such that $A'_U = A_U \cup \{(s_0, v) | v \in T\}$. This modification will not form new cycles in the resulted graph. Eq.(8.2) still holds. Construct new weights $W : A'_U \cup A_F \cup A_B \to \mathbb{Q}$ for each arc such that:

$$w(a) = \begin{cases} 0, & a(s_0, v), \\ M_0(a) - \delta^*, & \text{otherwise.} \end{cases} \tag{8.3}$$

According to Eq.(8.2)(8.3), no cycle has negative weights. Thus the shortest path between s_0 and $v \in T$ exists with respect to the weight defined in Eq.(8.3). Let this shortest path be $r(v)$ and $r(s_0) = 0$. For every arc $a(v_i, v_j) \in A_U \cup A_F \cup A_B$, the shortest path between s_0 and v_j should be no longer than the path which consists the shortest path between s_0 and v_j plus $a(v_i, v_j)$. This is equivalent to:

$$r(v_j) \leq r(v_i) + w(a) \Leftrightarrow r(v_j) - r(v_i) \leq M_0(a) - \delta^* \tag{8.4}$$

Thus $r(v)$ satisfies the requirements in the theorem. □.

Based on Theorem 20, a MILP formulation for optimal buffer resizing of a PBPG$< T, A, M_0, B >$ is given as:

Minimize:$\sum_{a \in A_F} B(a)$,

Subject to:

$$r(v_j) - r(v_i) \leq M_0(a) - \delta^* \tag{8.5}$$
$$r(v_i) - r(v_j) \leq B(a) - M_0(a) - \delta^*, \tag{8.6}$$
$$B(a) \geq 2, \tag{8.7}$$

where $a(v_i, v_j) \in A_F$,

$$r(v_j) - r(v_i) \leq M_0(a) - \delta^* \tag{8.8}$$

where $a(v_i, v_j) \in A_U$.

Considering the definition of PBPG, only the arcs in A_F have finite capacities and can be resized. Therefore, only these arcs appear in the objective function. Eq.(8.6) uses the relation $M_0(a') = B(a) - M_0(a)$ in Section 6.1. Reconstructing SCCG after MBPA leads to SCCG$_f(V_{sccf}, A_{sccf}, \delta_f, \delta'_f, MinV_f)$. Let scc_m be any node from V_{sccf} which is merged from SCC scc_0, scc_1, ..., scc_k in the original PBPG. One can directly apply above MILP formulation to each scc_m to make Assumption 1 true by setting $\delta^* = \min\{\delta(scc_i), i = 1, 2 \ldots, k$.

Doing so is sufficient to make SCCG$_f$'s throughput back to δ_U (the maximum throughput a PBPG can achieve), however it is not necessary. In fact, one only need to set $\delta^* = \delta_U$ and apply this MILP formulation to the whole graph. Since $\delta_U \leq \min\{\delta(scc_i), i = 1, 2 \ldots, k$, meeting $\delta^* = \delta_U$ would require less buffer size addition than directly making Assumption 1 true. On the other hand, the discussion about PBPG's boundness in Chapter 6 is based on Assumption 1, we also need to show that after MBPA, meeting $\delta^* = \delta_U$ is also enough to make a resulted PBPG bounded.

Theorem 21. *Let SCCG$_f(V_{sccf}, A_{sccf}, \delta_f, \delta'_f, MinV_f)$ be a resulted graph of SCCG$(V_{scc}, A_{scc}, \delta, \delta', MinV)$ after using MBPA, if for every direct cycle C of SCCG$_f$, $\delta(C) = \delta_U$, SCCG$_f$ is bounded.*

By Theorem 14, we only need to show that for every direct cycle C, if $\delta(C) = \delta_U$, $v \in V_{sccf}$ is reachable from $MinV_f$. We first prove this fact.

Lemma 4. *For the same description and condition stated in Theorem 21, $\forall v \in MinV$, suppose v is merged into $u \in V_{sccf}$ in SCCG$_f$, then $u \in MinV_f$.*

Proof: for every direct cycle C of SCCG$_f$, if $\delta(C) = \delta_U$, $\delta(MinV_f) = \delta_U$. Let v be any SCC from $MinV$ and u be corresponding SCC in SCCG$_f$ where v is merged into. We have $\delta(v) = \delta(MinV) = \delta_U$. On the other hand, we have $\delta(u) \geq (\delta_U)$ (critical cycle) and $\delta(u) \leq (\delta(v)) = \delta_U$, so $\delta(u) = \delta_U = \delta(MinV_f)$. Therefore, u belongs to $MinV_f$. □

With this lemma, we can direct prove Theorem 21.
Proof: Let u_0 be any node from V_{sccf} and v_0 be any sub SCC it merged from. After MBPA, v_0 is reachable from one SCC in $MinV$, denote it as v_x. Let $u_x \in V_{sccf}$ be the SCC in SCCG$_f$ which contains v_x. By previous Lemma, u_x is in $MinV_f$. Therefore, u_0 is reachable from u_x in SCCG$_f$. □

This theorem also further indicates Assumption 1 is not a necessary condition for the boundness of a PBPG, thus MBPA is a heuristic approach for back pressure minimization and may not get the global optimum for every case. Solving MILP formulation is NP hard and reducing the formulation size is preferred. In the following sections, we explores two approaches for MILP formulation reduction.

8.3 A reduced MILP (RMILP) formulation based on SCC level abstraction

As mentioned in Section 8.1, the throughput of system resulted from MBPA, may be less than δ_U. Because new cycles formed by added inter-SCC back pressure arcs may have a cycle-mean less than δ_U. Denote the set of the newly formed cycles as $NewC$. Let the resulted graph of MBPA be $SCCG'$ and its throughput be δ_{MBPA}, $\delta_{MBPA} \le \delta_U$ iff $\exists C \in NewC$ such that $\delta(C) = \delta_{MBPA} \le \delta_U$.

Note that the constraints expressed in Eq.(8.5)∼(8.8) guarantee that in the final system, any cycle C satisfies $\delta(C) \ge \delta_U$. However, this is overly restrictive. Because, by the definition of δ_U, any cycle C in the original graph system before MBPA (not in $NewC$), has already satisfied $\delta(C) \ge \delta_U$. Unnecessary restrictions are used in MILP formulation for these arcs not in $NewC$. This fact gives an opportunity to simplify the this MILP formulation by only constraining the cycle-mean on $NewC$. Some definitions are given to ease the discussion about the constitution of $C \in NewC$.

Definition 17 (interface). *In a $SCCG(V_{scc}, A_{scc}, \delta, MinV, \delta')$, the interface of a $scc_k \in V_{scc}$ is a set of transitions of scc_k such that $\forall v \in interface(scc_k)$, $\exists a(v, v') \in A_{scc}$, where v' is a transition of scc_j, a different SCC other than scc_k.*

$Interface(scc_k$ is the set of transitions in scc_k which are directly connected to other SCCs. In Fig. 8.1 and Fig. 8.2, $interface(v1) = \{T23, T21\}$, $interface(v2) = \{T20\}$, $interface(v3) = \{T8, T2, T4, T10\}$, $interface(v4) = \{T24\}$, $interface(v5) = \{T16, T18\}$.

In addition, we define the set of directed paths within scc_k whose two ends are both in $interface(scc_k)$ as $EPath(scc_k)$ in Definition 18.

Definition 18 (EPath). *$EPath(scc_k)$ is a set of directed paths in scc_k such that $\forall \pi \in EPath(scc_k)$ with a length $|\pi| = L$, $\pi = t_0 \to t_1 \to \ldots \to t_{L-1}$. t_k are transitions of scc_k, both t_0 and $t_{L-1} \in interface(scc_k)$.*

Let $\pi \in EPath(scc_k)$ and $\pi = v_0 \xrightarrow{a_0} v_1 \xrightarrow{a_1} \dots \xrightarrow{a_{N-1}} v_N$. Summing up $r(v_{i+1}) - r(v_i) \leq M_0(a_i) - \delta_U$ along π leads to:

$$r(v_N) - r(v_0) \leq \sum_{a_i \in \pi} M_0(a_i) - |\pi|\delta_U = M_0(\pi) - |\pi|\delta_U \tag{8.9}$$

Here, $M_0(\pi) = \sum_{a_i \in \pi} M_0(a_i)$, and π can be treated as an *abstracted arc* from v_0 to v_N with the initial tokens $M_0(\pi)$. Eq.(8.9) can be regarded as a constraint for path π. Note that any cycle $C \in NewC$ consists of paths in $EPath$ and inter-SCC arcs, intuitively we only need the constraints for arcs in $EPath$ and the inter-SCC arcs to make sure $\delta(C) \geq \delta_U$, where $C \in NewC$.

For a particular scc_k, only the constraints of $\pi \in EPath(scc_k)$ contribute to the restriction of $\delta(C)$. Denote $\Pi(v_s, v_e)$ as the set of paths in $EPath(scc_k)$ which is from v_s to v_e, where $v_s, v_e \in interface(scc_k)$. Therefore, we only require that:

$$\begin{array}{l} \forall v_s, v_e \in interface(scc_k), \forall \pi \in \Pi(v_s, v_e), \\ r(v_e) - r(v_s) \leq M_0(\pi) - |\pi|\delta_U \Leftrightarrow \\ r(v_s) - r(v_e) \leq \min_{\pi \in \Pi(v_s,v_e)}(M_0(\pi) - |\pi|\delta_U) \end{array} \tag{8.10}$$

Through this modification, only the transitions in $interface(scc_k)$ are kept in the constraints and other transitions inside the scc_k are abstracted. For any pair of $(v_s, v_e) \in interface(scc_k)$, the constraints of every path in $\Pi(v_s, v_e)$ can be represented by path $\pi(v_s, v_e)^* \in \Pi(v_s, v_e)$ which has the minimum $M_0(\pi) - |\pi|\delta_U$. Since $M_0(\pi) - |\pi|\delta_U$ are constants, $\pi(v_s, v_e)^*$ can be determined before solving the MILP formulation by finding the shortest path between (v_s, v_e) with the arc weight $M_0(a) - \delta_U$. Through this abstraction, only transitions in $interface(scc_k)$ and abstracted arcs $\pi^*(v_s, v_e)$ for every pair of transitions $(v_s, v_e) \in interface(scc_k)$ are kept.

Further reduction can be made based on the directions of inter-SCC arcs connected to scc_k. Transitions in $interface(scc_k)$ can be classified into three categories by defining a map $TP : interface(scc_k) \rightarrow \{IN, OUT, INOUT\}$. Formally,

- $TP(v) = IN$, iff $\exists a(u, v)$ where $u \in interface(scc_i)$ and $i \neq k$ but not $\exists a'(v, w)$ where $w \in interface(scc_j)$ and $j \neq k$.
- $TP(v) = OUT$, iff $\exists a(v, u)$ where $u \in interface(scc_i)$ and $i \neq k$ but not $\exists a'(w, v)$ where $w \in interface(scc_j)$ and $j \neq k$.
- $TP(v) = INOUT$, iff $\exists a(u, v)$ and $a'(v, w)$ where $u \in interface(scc_i)$ and $i \neq k$, $w \in interface(scc_j)$ and $j \neq k$.

For example, $v3$ in Fig. 8.1 has $TP(T8) = TP(T2) = INOUT$, $TP(T4) = In$ and $TP(T10) = OUT$. If $TP(v) = IN$, any paths in $EPath(scc_k)$ which terminate at v can't be in cycles $C \in NewC$,e.g. $(T8, T4)$, $(T2, T4)$ and $(T10, T4)$ can't be in new cycles. Therefore, arcs $\pi(u, v)^*$ are not useful for $\delta(C) \geq \delta_U$ and can be removed. Similarly, if $TP(v) = OUT$, arcs $\pi(v, w)^*$ can be removed. Denote there are n_{k1}, n_{k2} and n_{k3} transitions in $interface(scc_k)$ whose types are IN,

OUT and $INOUT$, where $n_{k1} + n_{k2} + n_{k3} = n_k$. Instead of $n_k(n_k - 1)$, $n_{k1}n_{k2} + n_{k1}n_{k3} + n_{k2}n_{k3} + n_{k3}n_{k3}$ arcs are necessary when constructing the MILP formulation with respect to scc_k.

In this way, scc_k can be abstracted as a directed graph $G_{scc_k}(V_a(scc_k), A_a(scc_k), M_{0a}(scc_k), D_a(scc_k))$, where $V_a(scc_k) = interface(scc_k)$, $A_a(scc_k) = \{(v_s, v_e)|v_s, v_e \in V_a(scc_k)\}$ and $M_0(v_s, v_e) = M_0(\pi(v_s, v_e)^*)$ and $d(v_s, v_e) \in D_a(scc_k)$ is the latency of (v_s, v_e), $d(v_s, v_e) = |\pi(v_s, v_e)^*|$.

One may notice that if $scc_k(V(scc_k), A(scc_k))$ is a sparse graph, $|A_a(scc_k)|$ might be large compared to the number of original arcs in scc_k, denoted as $|A(scc_k)|$. In this case, the formulation size reduced by $V_a(scc_k)$ is offset by the increasing complexity by $A_a(scc_k)$. We introduce a reduction $ratio = \frac{|A_a(scc_k)||V_a(scc_k)|}{|A(scc_k)||V(scc_k|}$, which represents the reduction we get through abstraction. The smaller the $ratio$, the more reduction we can achieve. A $threshold$ can be employed to determine whether this abstraction should be done. If $ratio > threshold$, we will keep scc_k in the final graph without abstraction. In our experiment, we set $threshold = 0.8$, considering the overhead of abstraction. $ratio$ can be computed before computing $\pi^*(v_s, v_e)$.

Now, we are ready to show the steps of obtaining a SCC level abstraction graph $G_a(V_a, A_{Ua}, A_{Fa}, A_{Ba}, M_{0a}, D_a, B_a)$ from the resulted graph G' of MBPA, G is the original PBPG $(T, A_U, A_F, A_B, M_0, B)$ without back pressure.

1. Construct $SCCG(V_{scc}, A_{scc}, \delta, MinV, \delta')$ of G and construct the abstracted graph $G_{scc_k}(V_a(scc_k), A_a(scc_k), M_{0a}(scc_k), D_a(scc_k))$ for each $scc_k \in V_{scc}$ as previously discussed.
$V_a = \bigcup_{scc_k \in V_{scc}} V_a(scc_k)$, $A_a = \bigcup_{scc_k \in V_{scc}} A_a(scc_k)$.
2. For the inter-SCC arc a, we just keep them in the abstracted graph. If $a \in A_U$, add a to A_{Ua}; if $a \in A_F$, add a to A_{Fa}; if $a \in A_B$, add a to A_{Ba}. The initial tokens and latency of a remains same in G_a as in G such that $M_0(a) = M_{0a}(a)$ and $d(a) = 1$. Thus $A_{Fa} = A_F$ and $A_{Ba} = A_B$.

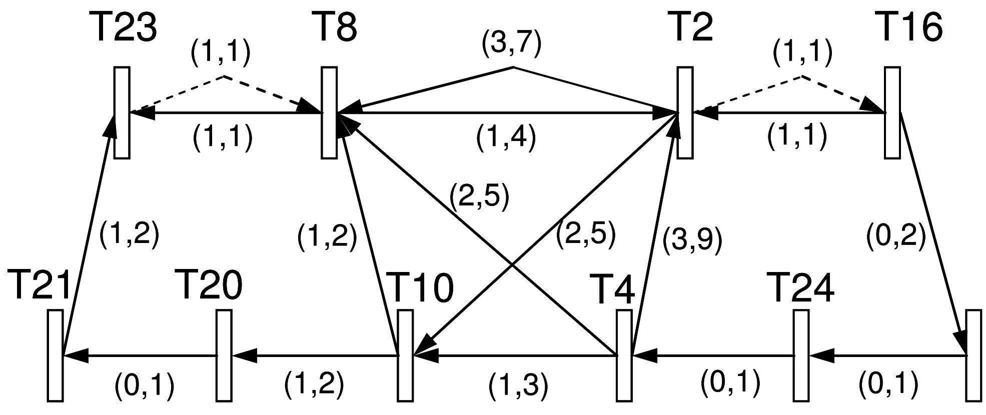


Figure 8.3: The reduced graph based on SCC level abstraction

The abstracted graph of Fig. 8.1 is shown in Fig. 8.3. For each arc a, $(M_0(a), D(a))$ is marked along the arc, e.g $M_0(T2, T8) = 3$, $d(T2, T8) = 7$. The reduced MILP formulation is given based on G_a and we refer it as RMILP in the rest of this chapter.

Minimize:$\sum_{a \in A_{Fa}} B(a)$,

Subject to:

$$r(v_j) - r(v_i) \leq M_0(a) - d(a) * \delta_U \tag{8.11}$$

$$r(v_i) - r(v_j) \leq B(a) - M_0(a) - d(a) * \delta_U, \tag{8.12}$$

$$B(a) \geq 2, \tag{8.13}$$

where $a(v_i, v_j) \in A_{Fa}$,

$$r(v_j) - r(v_i) \leq M_0(a) - d(a) * \delta_U \tag{8.14}$$

where $a(v_i, v_j) \in A_{Ua}$.

Finally, we need to prove that the RMILP formulation guarantees that each $C \in NewC$ satisfy $\delta(C) \geq \delta_U$ as described in Theorem 22.

Theorem 22. *For every $C \in NewC$, $\delta(C) \geq \delta_U$ iff exists a function $r : T_a \to \mathbb{R}$, such that $r(v_j) - r(v_i) \leq M_0(a) - d(a) * \delta_U$, where $a(v_i, v_j) \in A_{Ua} \cup A_{Fa} \cup A_{Ba}$.*

Proof: "if:" Let $C \in NewC$, C consists of inter-SCC arcs and paths going through SCCs. Suppose C is across scc_k through a path π_i from v_s to v_e, where $v_e, v_s \in interface(scc_k)$ and $\pi_i \in \Pi(v_s, v_e)$. Assume $r(v_e)$ and $r(v_s)$ exist for $r(v_e) - r(v_s) \leq M_0(a_a) - d(a_a)\delta_U$, where a_a is the arc from v_s to v_e in the abstracted graph G_a. Recall that a is the abstraction of π^*, the path in $\Pi(v_s, v_e)$ with the minimal $\{M_0(\pi) - d(\pi)\delta_U\}$ and $M_0(a) = M_0(\pi^*)$, $d(a) = d(\pi^*)$. Therefore, $M_0(a) - d(a)\delta_U \leq M_0(\pi) - d(\pi)\delta_U$ leading to $r(v_e) - r(v_s) \leq M_0(\pi) - d(\pi)\delta_U$. Summing up these inequalities along cycle C leads to $\delta(C) = \frac{\sum_{\pi \in C} M_0(\pi)}{\sum_{\pi \in C} d(\pi)} \geq \delta_U$.
"only if:" If every $C \in NewC$ satisfies $\delta(C) \geq \delta_U$, by the definition of δ_U all cycles in the system satisfies $\delta(C) \geq \delta_U$. We can use the same way as in proof of Theorem 20 to construct $r(v_i)$ such that $r(v_j) - r(v_i) \leq M_0(a_i) - \delta_U$ holds for every arc $a_i(v_i, v_j)$ in the original graph G. Every arc $a_a(v_s, v_e)$ in G_a is either a inter-SCC arc or an abstracted arc within one SCC. If a_a is inter-SCC arc, $r(v_e) - r(v_s) \leq M_0(a_a) - \delta_U$ already satisfied. If a_a is an abstracted arc in scc_k, then a_a relates to the path $\pi^*(v_e, v_s)$. $r(v_e) - r(v_s) \leq M_0(a_a) - \delta_U$ holds by adding these inequalities along π^*. $\square$

8.4 A localized MILP (LMILP) formulation for throughput improvement

Previous section reduce the MILP formulation by analyzing how newly critical cycles are formed. Another perspective for this problem is to "localize" the newly critical cycle to a small part of the resulting graph. After MBPA, added back pressure arcs merge some of the SCCs in the original SCCG into one bigger SCC. Therefore, we recalculate strongly connected component graph SCCG$_f(V_{sccf}, A_{sccf}, \delta_f, \delta'_f, MinV_f)$ by recomputing the SCCs. Since SCCG$_f$ is a DAG, all of the newly formed arcs must reside in these subgraphs which correspond to nodes in V_{sccf} including critical cycles whose cycle mean are less than δ_U. Throughput reduction of SCCG$_f$ is due to throughput reduction of some nodes in V_{sccf}. Again, these nodes should contain critical cycles. Therefore, we are able to localize newly critical cycles by checking whether $\delta(v), v \in V_{sccf} < \delta_U$. If this is true for a SCC $v \in V_{sccf}$, we call such v "problematic SCC". In another word, if we apply above formulation of Section 8.2 for each problematic scc_m by setting $\delta^* = \delta_U$, throughput of scc_m could be improved to δ_U. Again, the throughput of SCCG$_f$ is increased to δ_U, by solving MILP for these problematic SCCs.

In constrast to [56, 15] which apply the MILP formulation for the whole graph representation of the LIS, we use the MILP formulation only for the problematic SCCs. In many cases, this is only a small part of the original graph. We refer this method as *localized MILP* (LMILP) formulation. LMILP can also increase the throughput to δ_U but generally has a much smaller size since it is only applied to subgraphs (the problematic SCCs) of the PBPG. On the contrary to RMILP, LMILP does not have the disadvantage of RMILP that more constraints might be brought in each SCC is a sparse graph. That's why a threshold is used to determine whether a particular SCC should be abstracted or not(see the discussion in Section 8.3). LMILP, by its name, localizes the new critical cycle to several SCCs. LMILP is likely to be more efficient than RMILP. More detailed comparison is done in Section 8.5.

8.5 Experimental results

As presented previously, we can first utilize the MBPA to add minimum number of back pressure arcs to make the system bounded and then construct the RMILP or LMILP formulation to increase its throughput back to δ_U if $\delta_{MBPA} < \delta_U$. Then RMILP or LMILP formulation is applied to increase its throughput. The combined approach has the following benefits in optimizing the system's performance:

- The number of back pressure arcs is minimized;
- The throughput of the system is optimized to its upper bound δ_U;
- The approach has efficient solution;

- The required buffer size is small.

We have already covered the first two points in Section 6.3, Section 8.3 and Section 8.4. Now, let's analyze the complexity of our approach. We refer our approach which first uses MBPA then LMILP as LMILP$_{\text{MBPA}}$, the approach of ([56]) which apply the MILP formulation for an LIS where all the arcs have back pressures as MILP$_{\text{HS}}$, and the approach which uses RMILP for the results of MBPA as RMILP$_{\text{MBPA}}$. A group of ISCAS'89 and ITC benchmark which suffer throughput reduction after MBPA are chosen to compare the complexity of the three approaches and the results are shown in Table 8.1. There might be more than one test case for the same benchmark, which are obtained with different latencies. They are distinguished by subscripts such as $s349_0$ and $s349_1$. Column δ_U shows the maximum achievable throughput. Column δ_{HS} and δ_{MBPA} show the throughput with or without back pressure optimization. Values in these two columns are less than that in column δ_U for the same test case, which indicates that all three approaches are required to solve their MILP formulations to achieve δ_U. Since solving MILP is NP-hard, the size of MILP formulation of each approach predominates its performance.

We compare the three MILP formulations' sizes by the numbers of integer variables, fractional variables, and constraints.

1. Integer variables: The number of integer variables of MILP formulation is the same as the number of back pressure arcs of the PBPG. Therefore, LMILP$_{\text{MBPA}}$ and RMILP$_{\text{MBPA}}$ with back pressure minimization should have much less integer variables compared to MILP$_{\text{HS}}$ as shown in column $|I|$ of the corresponding approach. In addition, LMILP$_{\text{MBPA}}$ only applied to a subgraph of the graph to which RMILP$_{\text{MBPA}}$ applied, so the integer variable of LMILP is no more than that of RMILP.

2. Fractional variables and constraints: For most cases, the fractional variables and constraints of RMILP$_{\text{MBPA}}$ and LMILP$_{\text{MBPA}}$ are also less than that of MILP$_{\text{HS}}$ as shown in columns $|F|$ and $|cst|$, since both of them have reduction of the MILP's size by either doing an abstraction or only considering a subgraph. For test cases with smaller sizes, the complexity difference between RMILP$_{\text{MBPA}}$ and LMILP$_{\text{MBPA}}$ are not obvious. However, the number of constraints in the formulation of RMILP$_{\text{MBPA}}$ will increase dramatically if the corresponding PBPG is a sparse graph. That's why a "threshold" is introduced to determine whether a SCC should be abstracted or not. So LMILP$_{\text{MBPA}}$ demonstrates a smaller size in its MILP formulation when the test case is big.

The execution time of the three approach are shown in colum $t1$ (MILP$_{\text{HS}}$), $t2$ (RMILP$_{\text{MBPA}}$) and $t3$ (LMILP$_{\text{MBPA}}$). For most cases, we have $t3 > t2 > t1$. When the PBPG becomes larger, MILP$_{\text{HS}}$ does not converge and aborted because of exhausted memory consumption after 30 minutes execution, as indicated by $> 30M$. Above all, LMILP has the best performance among these three approaches.

Finally, we compare the buffer size between LMILP$_{\text{MBPA}}$ and MILP$_{\text{HS}}$ in Table 6.1. For MILP$_{\text{HS}}$, each arc has back pressure and requires at least 2 buffer sizes [16]. So there are at least $2 * |A|$

Table 8.1: Experiment results on throughput improvement

Bench	Throughput Info.			MILP$_{\text{HS}}$			RMILP$_{\text{MBPA}}$			LMILP$_{\text{MBPA}}$			Exec.(s)		
Mark	δ_{HS}	δ_{MBPA}	δ_U	$\lvert I\rvert$	$\lvert F\rvert$	$\lvert cst\rvert$	$\lvert I\rvert$	$\lvert F\rvert$	$\lvert cst\rvert$	$\lvert I\rvert$	$\lvert F\rvert$	$\lvert cst\rvert$	t1	t2	t3
$b10$	0.30	0.34	0.38	669	499	1338	19	176	943	19	410	566	0.89	0.56	0.23
$s349_0$	0.74	0.74	0.79	320	217	640	13	192	301	13	200	313	0.22	0.08	0.08
$s349_1$	0.56	0.56	0.57	443	340	886	19	158	584	19	315	434	0.42	0.22	0.16
$s382$	0.67	0.78	0.8	358	213	716	10	143	278	6	117	205	2.77	0.09	0.09
$s386$	0.42	0.52	0.53	586	405	1172	13	234	762	9	274	405	$> 30M$	0.38	0.20
$s400$	0.67	0.71	0.77	375	220	750	10	145	284	7	125	221	3.24	0.16	0.08
$s444$	0.67	0.73	0.8	405	237	810	11	137	277	7	147	256	119.7	0.11	0.09
$s713_0$	0.63	0.73	0.75	731	568	1462	70	317	1604	70	568	574	$> 30M$	0.86	0.33
$s713_1$	0.32	0.35	0.36	1366	1203	2732	113	609	1999	71	800	998	4.09	2.66	1.08
$s820_0$	0.33	0.35	0.36	1566	1116	3132	42	620	3869	42	806	1156	35.8	2.98	1.03
$s820_1$	0.29	0.31	0.32	1689	1239	3378	38	700	3625	38	890	1241	14.2	3.35	1.30
$s832$	0.32	0.33	0.34	1914	1450	3828	41	773	3687	41	1045	1408	7.41	4.33	1.81
$s1488_0$	0.53	0.53	0.54	2198	1463	4396	10	922	3856	10	729	1114	9.95	3.16	1.05
$s1488_1$	0.39	0.39	0.41	2358	1632	4716	23	1011	4770	23	824	1212	13.2	4.00	1.27
$s1488_2$	0.32	0.32	0.33	3445	2719	6890	12	1608	4718	12	1379	1759	$> 30M$	8.95	2.84
$s1488_3$	0.32	0.36	0.37	2844	2118	5688	11	1268	4261	11	1087	1469	$> 30M$	6.50	2.02
$s1488_4$	0.27	0.33	0.34	3000	2274	6000	11	1359	4352	11	1163	1541	$> 30M$	7.75	2.47
$s1494$	0.51	0.51	0.52	2450	1712	4900	15	1054	4287	15	895	1301	15.7	4.80	1.53

buffer size required as shown in column B$_{\text{HS}}$. On the other hand, the buffer size of LMILP$_{\text{MBPA}}$ can be computed according to Chapter 7. Generally, most arcs have $B(a) = 1$. So the total buffer size is expected to be smaller. We further compare the buffer size requirement when the system is initialized from S_u or S_v as discussed in Section 7.4. The required buffer sizes are shown in columns B_{Su} and B_{Sv}. For all cases, $B_{\text{HS}} > B_{Su} > B_{Sv}$.

Chapter 9

Conclusion and Future work

Latency insensitive design provides a synchronous solution for composing IP modules in SOC industry. With the proper design of wrappers and relay stations, different IP modules can be made to overcome wiring latencies in SOC design.

Composition of IP modules in LIP design can potentially cause deadlock and introduce communication mistakes due to interface incompatibilities. In Chapter 4, we show that SIGNAL language and its formal model polychrony can be employed to specify and verify these properties related to composition by static analysis. It avoids state space generation based verification such as model checking. With the SIGNAL syntax, the elementary hardware combinational logic, sequential logic and FSM can be correctly described, based on which more complicated LIP can be expressed. On the other hand, formal structures SDFG and CH constructed from SIGNAL description, can be used to analyze endo-isochrony and therefore verify the correct communication. These facts provide us an appropriate framework to model and verify the communication of LIP and which is demonstrated and validated through a case study.

In past, most LIS implementations use hand shake for every inter-IP connection to make IP blocks elastic to wiring latencies. However, handling back pressure requires relatively complicated logics and consumes more area and power. Also, it introduces unnecessary stalls for the pipeline which could possibly reduce the throughput. In addition, throughput optimization technique solves a MILP formulation where the integer variable numbers equals back pressure numbers which does not scale for big systems. Reducing handshake provides an opportunities for LIS optimization.

A novel design flow for LIS optimization based on back pressure minimization is discussed in Chapter 5, 6, 7, 8. Observing that using back pressure for every communication link is unnecessary, we model an LIS using PBPG and derived a sufficient and necessary condition with which boundness is kept for any system with multiple SCCs in Chapter 6. Based on this condition, MBPA is derived to add minimum number of back pressure arc to the system.

Back pressure may form new critical cycles along which data flow suffers a throughput reduction. Chapter 8 proposes two formulations: RMILP and LMILP for throughput improvement by buffer resizing. RMILP simplifies MILP formulation based on SCC-level abstraction, while LMILP localize new critical cycle within problematic SCCs. Both approaches can optimize system's throughput and are more scalable than conventional techniques. LMILP, in most case, outperforms RMILP and other techniques.

After MBPA and throughput improvement steps, LIS is bounded and has periodic schedulings. In Chapter 5, we have shown how the schedulings of LIS can be analyzed within the framework of periodic clock calculus. By modeling the scheduled latency insensitive protocols with marked graph, we derive the relations of scheduling sequences for different transitions and express such relations using a proposed initial scheduling matrix. Also, we provide a static analysis of the number of buffers required at a reconvergent block where data are accumulated due to barrier synchronization.

In the end, an implementation of the resulted LIS is proposed which refines its PBPG model. The implementation avoids using shift registers to store scheduling sequences for each IP module, which saves area and reduces power consumption.

9.1 Future work

The formal analysis and verification of globally asynchronous and locally synchronous design done in this dissertation can be extended in the following directions:

1. Work presented in Chapter 6 assumes that the latencies and execution clock cycles of each IP block are known. In this case, throughput of each SCC is statically determined before adding back pressure. However, this is not enough to analyze IP blocks which have variable latencies. IP block such as caches may have different latencies because there can be both a cache hit and miss for cache operation. We could extend the SCCG to involve SCCs whose throughputs are unknown or within a range. Based on this, boundness analysis and back pressure computation should be reconsidered. With this work, one can analyze and optimize LIS for a larger variety of SOC design which is more useful for industrial applications.

2. Our flow for LIS design optimization can also be extended to GALS software systems where multi-threads are executing on different rates thus have distinct throughput. Communication between threads requires local memory for data synchronization. It is necessary to validate weather the communication mechanism is bounded for the system such that threads will not overflow their local memory and corrupt the system. Also, our flow can be extended to check whether the synchronization or handshakes among threads are necessary and find minimum synchronization and local memory. It requires a novel approach to quantify "throughput" for software thread which is different from synchronous hardware.

3. Instead of ASAP scheduling, other schedulings can be employed for the LIS design. One promising application is for peak power optimization. Works have been done to heuristically

schedule operations at a system level to optimize SOC's peak power. We can employ a similar method for scheduled LIS design. One needs to first obtain the power profile of each IP block and apply these existing algorithms to find schedules. Also, as we need to implement such scheduling sequences for each IP block, the implementation should be decentralized and avoid using shift registers.

4. In this dissertation, we have tried using the polychrony framework to verify the communication of latency insensitive designs. This can also be extended to analyze and verify communication for IPs with asynchronous interfaces. Other works include using affine clock relation to specify the "rates" of inputs and outputs and further express throughput to analyze the composition of modules represented by SIGNAL.

Bibliography

[1] http://edmonds-alg.sourceforge.net/.

[2] A. Agiwal and M. Singh. Multi-clock latency-insensitive architecture and wrapper synthesis. *Electron. Notes Theor. Comput. Sci.*, 146(2):5–28, 2006.

[3] P. Amagbégnon, L. Besnard, and P. L. Guernic. Implementation of the data-flow synchronous language signal. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 163–173, New York, NY, USA, 1995. ACM.

[4] P. Aubry, P. L. Guernic, and S. Machard. Synchronous distribution of SIGNAL programs. In *Proc. of HICSS-29*, volume 1, pages 656–665, January 1996.

[5] E. Beigne and P. Vivet. Design of on-chip and off-chip interfaces for a gals noc architecture. In *ASYNC '06: Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, page 172, Washington, DC, USA, 2006. IEEE Computer Society.

[6] A. Benveniste. Some synchronization issues when designing embedded systems from components. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 32–49, London, UK, 2001. Springer-Verlag.

[7] A. Benveniste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. In *Proc. of Concurrency Theory*, 1664, pages 162–177. Springer, 1999.

[8] A. Benveniste, B. Caillaud, and P. L. Guernic. Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation. *Information and Computation*, 163(1):125–171, 2000.

[9] Bernard. The synchronous programming language signal a tutorial. Espresso project, HOUSSAIS IRISA, September 2004.

[10] F. Bock. An algorithm to construct a minimum directed spanning tree in a directed network. *Developments in Operations Research, Gordon and Breach*, pages 29–44, 1971.

[11] J. Boucaron, R. de Simone, and J.-V. Millo. Latency-insensitive design and central repetitive scheduling. pages 175–183, July 2006.

[12] J. Boucaron, R. de Simone, and J.-V. Millo. Formal methods for scheduling of latency-insensitive designs. *EURASIP J. Embedded Syst.*, 2007(1):8–8, 2007.

[13] J. Boucaron, J.-V. Millo, and R. De Simone. Another glance at relay stations in latency-insensitive designs. 2005.

[14] F. R. Boyer, E. M. Aboulhamid, Y. Savaria, and M. Boyer. Optimal design of synchronous circuits using software pipelining techniques. *ACM Transactions on Design Automation of Electronic Systems*, 7:516–532, 1998.

[15] D. Bufistov, J. Júlvez, and J. Cortadella. Performance optimization of elastic systems using buffer resizing and buffer insertion. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 442–448, Piscataway, NJ, USA, 2008. IEEE Press.

[16] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, 20(issue:9):1059–1076, Sep 2001.

[17] L. P. Carloni. The role of back-pressure in implementing latency-insensitive systems. *Electr. Notes Theor. Comput. Sci.*, 146(2):61–80, 2006.

[18] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic circuits. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(10):1437–1455, 2009.

[19] J. Carmona, J. Julvez, J. Cortadella, and M. Kishinevsky. Scheduling synchronous elastic designs. In *In Proc. 9th International Conference on Application of Concurrency to System Design (ACSD)*, 2009.

[20] M. R. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 576–581, New York, NY, USA, 2004. ACM.

[21] M. R. Casu and L. Macchiarulo. Adaptive latency-insensitive protocols. *IEEE Design and Test of Computers*, 24:442–452, 2007.

[22] A. Chakraborty and M. R. Greenstreet. Efficient self-timed interfaces for crossing clock domains. In *In Proceedings. 9th International Symposium on Asynchronous Circuits and Systems*, pages 78–88, 2003.

[23] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Trans. Comput.*, 22(4):421–422, 1973.

[24] D. M. Chapiro. *Globally-asynchronous locally-synchronous systems (performance, reliability, digital)*. PhD thesis, Stanford, CA, USA, 1985.

[25] T. Chelcea and S. M. Nowick. Robust interfaces for mixed-timing systems. *IEEE Trans. Very Large Scale Integr. Syst.*, 12(8):857–873, 2004.

[26] T. Chu. Synthesis of self-timed vlsi circuits from graph-theoretic specifications. Technical report, Cambridge, MA, USA, 1987.

[27] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

[28] A. Cohen, M. Duranton, C. Eisenbeis, and C. Pagetti. N-synchronous kahn networks - a relaxed model of synchrony for real-time systems. In *ACM International Conference on Principles of Programming Languages (POPL06)*, pages 180–193. ACM Press, 2006.

[29] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. Synchronization of periodic clocks. In *In ACM Conf. on Embedded Software (EMSOFT05)*, 2005.

[30] A. Cohen, L. Mandel, F. Plateau, and M. Pouzet. Abstraction of clocks in synchronous data-flow systems. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 237–254, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] R. Collins and L. Carloni. Topology-based performance analysis and optimization of latency-insensitive systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(12):2277–2290, Dec. 2008.

[32] J. Cortadella and M. Kishinevsky. Synchronous elastic circuits with early evaluation and token counterflow. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 416–419, June 2007.

[33] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 657–662, New York, NY, USA, 2006. ACM.

[34] A. Dasdan and R. K. Gupta. Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:889–899, 1997.

[35] S. Dasgupta and A. Yakovlev. Modeling and verification of globally asynchronous and locally synchronous ring architectures. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 568–569, Washington, DC, USA, 2005. IEEE Computer Society.

[36] R. Dobkin, T. Kapshitz, S. Flur, and R. Ginosar. Assertion based verification of multiple-clock gals systems. Technical report, VLSI Systems Research Center, Technion Israel Institute of Technology, June 2008.

[37] J. Edmonds. Optimum branchings. *J. Res. Nat. Bur. Standards,*, 71B:233–240, 1967.

[38] ESPRESSO Project, IRISA. The Polychrony Toolset. http://www.irisa.fr/espresso/Polychrony.

[39] R. Govindarajan and G. R. Gao. A novel framework for multi-rate scheduling in dsp applications, 1993.

[40] M. R. Greenstreet. Implementing a stari chip. In *IN PROC. INTERNATIONAL CONF. COMPUTER DESIGN (ICCD*, pages 38–43. IEEE Computer Society Press, 1995.

[41] P. L. Guernic, M. Borgue, T. Gauthier, and C. Marie. Programming real time applications with SIGNAL. *Proc. of the IEEE*, 79(9):1321–1335, 1991.

[42] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers. Synchronous interlocked pipelines. In *In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12. IEEE Computer Society Press, 2002.

[43] B. A. Jose, B. Xue, and S. K. Shukla. An analysis of the composition of synchronous systems. In *Formal Methods for Globally Asynchronous Locally Synchronous Design*, April 2009.

[44] G. Kahn. The Semantics of a Simple Language for Parallel Programming. *Proceedings of Information Processing*, pages 471–475, 1974.

[45] H. K. Kapoor. A process algebraic view of latency-insensitive systems. *IEEE Transactions on Computers*, 58:931–944, 2009.

[46] H. K. Kapoor and M. B. Josephs. Modelling and verification of delay-insensitive circuits using ccs and the concurrency workbench, 2003.

[47] M. Kishinevsky, S. K. Shukla, and K. S. Stevens. Guest editors' introduction: Gals design and validation. *IEEE Design and Test of Computers*, 24:414–416, 2007.

[48] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[49] W. J. Knottenbelt and J. T. Bradley. Tackling Large State Spaces in Performance Modelling. In *Formal Methods for Performance Evaluation*, volume 4486 of *Lecture Notes in Computer Science*, pages 318–370. Springer, June 2007. Bernardo, Hillston (Eds).

[50] X. Kong, R. Negulescu, and L. W. Ying. Refinement-based formal verification of asynchronous wrappers for independently clocked domains in systems on chip. In *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 370–385, London, UK, 2001. Springer-Verlag.

[51] S. R. Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 267–281, New York, NY, USA, 1982. ACM.

[52] M. Krstic, E. Grass, F. K. Gurkaynak, and P. Vivet. Globally asynchronous, locally synchronous circuits: Overview and outlook. *IEEE Design and Test of Computers*, 24:430–441, 2007.

[53] E. Lawvere. Combinatorial optimization: Networks and matroids. New York, NY, USA, 1976. Holt Rinehart and Winston Ed.

[54] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, pages 5–35, 1991.

[55] C.-H. Li and L. P. Carloni. Leveraging local intracore information to increase global performance in block-based design of systems-on-chip. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(2):165–178, 2009.

[56] R. Lu and C.-K. Koh. Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 227, Washington, DC, USA, 2003. IEEE Computer Society.

[57] R. Lu and C.-K. Koh. Performance analysis of latency-insensitive systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(3):469–483, March 2006.

[58] O. Maffeis and P. L. Guernic. Distributed Implementation of Signal: Scheduling & Graph Clustering. pages 547–566. Springer-Verlag, 3rd International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, September 1994.

[59] E. W. Mayr. An algorithm for the general petri net reachability problem. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 238–246, New York, NY, USA, 1981. ACM.

[60] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[61] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[62] D. Nowak. Synchronous structures. *Inf. Comput.*, 204(8):1295–1324, 2006.

[63] R. Lipton. The Reachability Problem Requires Exponential Space Technical Report(62). Yale University 1976.

[64] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[65] L. Y. Rosenblum and A. Yakovlev. Signal graphs: From self-timed to timed ones. In *International Workshop on Timed Petri Nets*, pages 199–206, Washington, DC, USA, 1985. IEEE Computer Society.

[66] G. Salaun, W. Serwe, Y. Thonnart, and P. Vivet. Formal verification of chp specifications with cadp illustration on an asynchronous network-on-chip. In *ASYNC '07: Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 73–82, Washington, DC, USA, 2007. IEEE Computer Society.

[67] K. Schmidt. LoLA: A Low Level Analyser. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000)*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474. Springer-Verlag, June 2000.

[68] J. N. Seizovic. Pipeline synchronization. In *In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 87–96. IEEE Computer Society Press, 1994.

[69] K. S. Stevens. Practical verification and synthesis of low latency asynchronous systems, 1994.

[70] S. Suhaib, B. A. Jose, S. K. Shukla, and D. Mathaikutty. Formal transformation of a kpn specification to a gals implementation. In *FDL*, pages 84–89, 2008.

[71] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla. Validating families of latency insensitive protocols. *IEEE Transactions on Computers*, 55(11):1391–1401, 2006.

[72] J.-P. Talpin, J. Ouy, L. Besnard, and P. L. Guernic. Compositional design of isochronous systems. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 928–933, New York, NY, USA, 2008. ACM.

[73] J.-P. Talpin, D. Potop-Butucaru, J. Ouy, and B. Caillaud. From multi-clocked synchronous processes to latency-insensitive modules. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 282–285, New York, NY, USA, 2005. ACM.

[74] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[75] R. E. Tarjan. Finding optimum branchings. *Networks*, 7:25–35, 1977.

[76] P. Teehan, M. Greenstreet, and G. Lemieux. A survey and taxonomy of gals design styles. *IEEE Design and Test of Computers*, 24:418–428, 2007.

[77] M. Vijayaraghavan and A. Arvind. Bounded dataflow networks and latency-insensitive circuits. In *MEMOCODE'09: Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign*, pages 171–180, Piscataway, NJ, USA, 2009. IEEE Press.

[78] B. Xue and S. K. Shukla. Analysis of Scheduled Latency Insensitive Systems with Periodic Clock Calculus. In *IEEE High Level Design Validation and Test Workshop (HLDVT09)*, San Francisco, CA, USA, 2009.

[79] A. V. Yakovlev, A. M. Koelmans, A. Semenov, and D. J. Kinniment. Modelling, analysis and synthesis of asynchronous control circuits using petri nets. *INTEGRATION: the VLSI Journal*, 21:143–170, 1996.

[80] J. You, Y. Xu, H. Han, and K. S. Stevens. Performance evaluation of elastic gals interfaces and network fabric. *Electron. Notes Theor. Comput. Sci.*, 200(1):17–32, 2008.

[81] K. Y. Yun. Synthesis of asynchronous controllers for heterogeneous systems, 1994.

[82] K. Y. Yun and R. P. Donohue. Pausible clocking: A first step toward heterogeneous systems. In *In Proc. International Conf. Computer Design (ICCD*, pages 118–123. IEEE Computer Society Press, 1996.