Pipeline Optimization for Asynchronous Circuits

by

Sangyun Kim

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Electrical Engineering)

March 2003

Dedication

This dissertation is dedicated to . . .

Acknowledgements

I would like to thank . . .

Contents

List Of Tables

List Of Figures

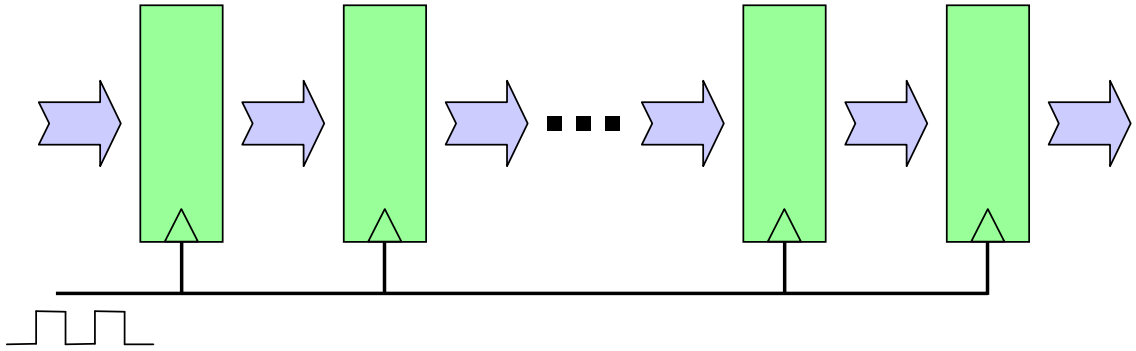
Abstract

This thesis is consist of two parts. The first part addresses computer-aided-design techniques to automate pipeline optimization of asynchronous circuits. The general problem is defined as identifying the minimal pipelining needed in an asynchronous circuit (e.g., number/size of pipeline stages/latches required) to satisfy a given performance constraint, thereby implicitly minimizing area and power for a given performance. In our work, we have proven that this problem is NP-complete and developed an efficient branch and bound solution that can find the optimal pipeline configuration for moderate sized deterministic pipelines models. The second part introduces structural reductions of probabilistic timed Petri nets that preserve a large class of performance measurements. In particular, we proposes a class of reductions that preserve efficiently computable bounds of statistics of time-separation of events (TSEs). It identifies two specific reductions within this class. It demonstrates the utility of these reductions by reducing a detailed Petri net describing the four-phase protocol of a well-known asynchronous pipeline template into a simpler two-phase architectural-level Petri net model. The benefit of this reduced model is that the run-time of subsequent TSE analysis can be greatly improved.

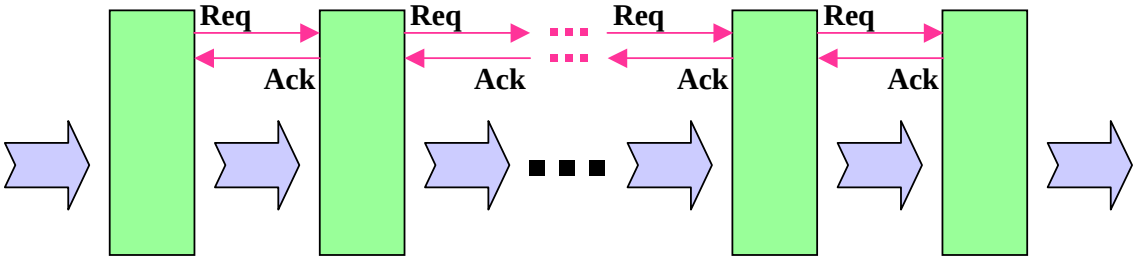
Chapter 1

Introduction

Correct operation of digital VLSI circuits requires proper synchronization of data flow. In traditional *synchronous circuits*, the circuit is divided into discrete combinational blocks each of which performs its operation within the *cycle time* of the periodic global clock. Thus, synchronous circuit designer must ensure that every combinational block can complete its operation within the cycle time under *all* possible circumstances, including all possible input combinations as well as worst case power supply and chip temperature. In this way, the global clock ensures synchronization among combinational blocks by guaranteeing that the output of every combinational blocks is valid and ready to consume before the next clock period begins. Recently, however, circuits that lack a global clock, called *asynchronous circuits*, have demonstrated potential benefits in low power, high average performance, composability, and improved noise immunity and electromagnetic compatability (e.g. [41, 31, 8, 66, 50, 78, 7, 28, 90, 6]). In contrast to synchronous circuits, asynchronous circuits are typically divided into functional units that communicate



(a) A synchronous circuit



(b) An asynchronous circuit

Figure 1.1: Synchronous circuits vs. asynchronous circuits

using handshaking-based message passing. Figure 1.1 illustrates the difference between typical synchronous and asynchronous circuits.

Numerous tools and techniques have been developed to help automate synchronous circuit design at various levels of abstraction [67, 33]. Some analogous tools have been designed for asynchronous circuit design in the areas of transistor sizing [14], technology mapping [17], hazard-freedom and area minimization logic synthesis [39, 20, 79]. In addition, some researchers have begun to address the allocation and scheduling [4, 3, 1] problem associated with high-level synthesis. Despite these works there exists no public domain design flow that can support both very high-performance circuits and automated design exploration at the various levels

of abstraction. In particular, performance estimation and optimization remains somewhat of a stumbling block in many levels of abstraction. The basic problem is that the complex interaction of various handshaking protocols often makes direct optimization for performance very difficult makes simulation as the only viable alternative.

1.1 Asynchronous Circuit Design Flow

The USC Asynchronous CAD and VLSI group, jointly with the Columbia Asynchronous group, is currently developing a complete asynchronous circuit design methodology that will support automated design exploration of both high-performance and low-power asynchronous circuits. The basic steps of this methodology are illustrated in Figure 1.2. First, a language based model, such as CSP [49] and Verilog [63], is used as the input description. This input description describes the desired top-level functionality of the chip and may be annotated with overall constraints on power, energy consumption, throughput, latency, chip area, etc. Note that details regarding internal structure or the specific asynchronous protocols used are specifically not included in the description. After generating this input description and verifying its correctness, the next step in the methodology is to explore and finalize a basic architecture for the design. This basic architecture should identify the number and relative characteristics of the basic blocks in the design (register files, ALUs, multipliers, etc.) To automate this step, we expect to adapt

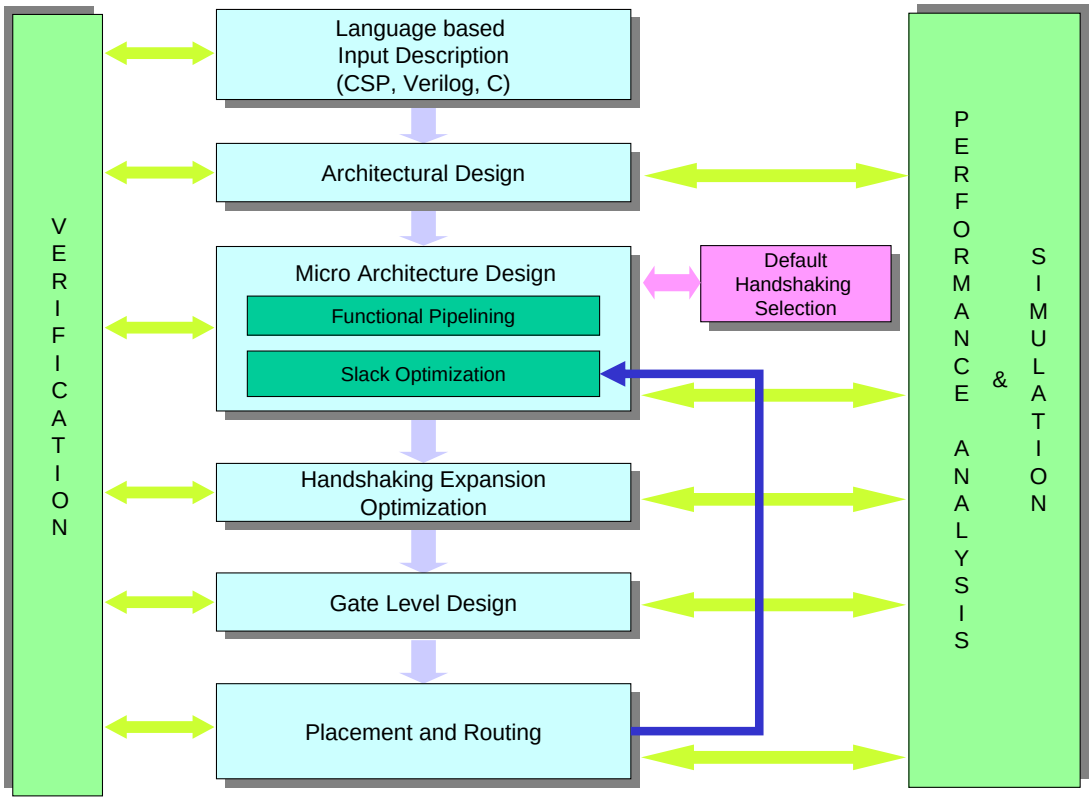


Figure 1.2: Asynchronous circuit design flow under development

variations in classical high-level synthesis, i.e., *scheduling, resource sharing, and binding*. Most likely, however, many practical designs will have a few architectural candidates that can be easily identified by the designer. After architectural design is complete, the next step in the methodology is micro-architecture design. This design consists of defining the level of functional pipelining of each basic unit and a step called *slack* optimization in which pipeline buffers are inserted in between pipeline stages. To better automate pipeline and slack optimization, we advocate creating a performance model of each pipeline stage and using analytical performance analysis techniques to guide the optimization tools that are the focus of this thesis. Once this initial micro-architecture is created, the next (optional) step is to identify critical components and perform *handshaking optimization* to achieve higher performance and lower power. This optimization often requires local timing assumptions to ensure correctness and thereby provides the designer with a trade-off between robustness, ease of timing validation and higher performance, lower power. Based on the final micro-architecture, a gate or transistor level design is generated. This step can be done either automatically using new template-based synthesis techniques that our group is creating or manually. Finally, placement and routing will be applied very similarly to that required in synchronous circuit design. In every step of the design process, verification and performance analysis tools are used to verify correct functionality and overall performance.

The focus of this proposal is performance analysis and optimization. Currently there are two basic approaches followed. The first involves using performance analysis techniques to guide manual or semi-automated design changes (e.g., [74]). This approach relies on sophisticated performance analysis techniques developed in [23, 16, 37, 88, 87]. The alternative approach is to develop synthesis techniques that directly optimize for performance. Successful efforts in this area have addressed transistor sizing [14], technology mapping [17], and allocation and scheduling (e.g., [4, 3, 1]). These techniques often use a more abstract, coarse model of performance and thus lack the precision of the former models but have the advantage of avoiding the long running time of many iterations of the analyze-modify loop. The proposed research is to attack a specific sub-problem of performance analysis and optimization called pipeline optimization and we expect our solutions will use aspects of both approaches.

1.2 Pipeline Optimization for Asynchronous Circuits

While it is well-known that good pipelining design styles in asynchronous circuits are critical to reduce the asynchronous control circuit overhead (e.g., [84, 74]), it is more difficult to determine the best means of breaking up a large combinational block into pipeline stages and the addition of pipeline buffers between stages to achieve a given performance. In fact, recent experiences suggest this optimization problem is getting more difficult. Namely, Caltech researchers, et al., proposed

partitioning asynchronous datapaths into bit-slices and pipelining between bit-slices to achieve higher throughput [50, 21]. When combined with standard pipelining between functional component boundaries, this creates a complex 2-dimensional pipeline. As a general rule in asynchronous design, the number of pipeline stages increases the power and area of the design due to extra completion sensing and control logic. Thus, one reasonable objective for pipeline optimization is to identify the minimal pipelining needed to satisfy a given performance constraint, thereby implicitly minimizing area and power for a given performance.

It is worth pointing out similarities of this problem with a somewhat analogous problem of *retiming* [69, 44] in the domain of synchronous circuits. In particular, like our problem, one basic version of retiming is to achieve a desired cycle time with the minimum number of latches. In addition, like retiming, we do not significantly change the structure of the circuit. That is, we currently *do not* consider re-synthesizing the circuit when we perform pipeline optimization. The key difference between the two problems, however, is that in the synchronous domain an initial assignment of latches must be given and the number of latches along any cycle must not be changed. In contrast, for our problem, the initial latch assignment is not necessary and the correctness requirements on the number of latches along a cycle are different. For example, increasing the number of latches in a cycle is often allowed and sometimes can increase throughput.

1.3 Reducing Probabilistic Timed Petri Nets

As previously mentioned, to support automatic pipeline and slack optimization, analytical performance models and performance analysis are both required. However, estimation and optimization of asynchronous circuit's performance, however, remains somewhat of a stumbling block due to the complex interaction of various handshaking protocols. Traditionally, performance analysis has been limited to simulation of detailed design models which suffers from long run-times and results that are subject to the reliability of the input vector statistics. For this reason, many analytical methods to analyze the performance of asynchronous circuits efficiently have been proposed [89]. Many of these techniques use timed Petri net with fixed delay [14], interval delay [23, 36, 55], and/or stochastic delay [88, 86] as circuit models. Numerous methods of analyzing these Petri nets have been proposed but most are limited to a small subclass of Petri nets or suffer from severe capacity limitations. For a large class of stochastic Petri nets (ones that do not involve arbitrarion), a recently developed approach using statistical simulation to obtain bounds on TSE statistics has been shown to be computationally efficient, able to handle models with hundreds of places and transitions. Nevertheless, real system models may have tens of thousands of places and transitions and increasing the capacity of analysis and reducing run-time is very important, particularly if analysis is used internal to a synthesis or design loop. It is this problem that this work addresses.

The goal of this work is to introduce simple reductions of Petri nets that preserve performance properties. In particular, we identify structural reductions that preserve bounds on TSE statistics, as computed using the algorithms in [88, 86]. This means that the reduced Petri nets can be analyzed instead of their more complicated counterparts. One intended use of this reduction algorithm is to simplify the performance model of small asynchronous cells which make up the building blocks of large asynchronous systems. These cells may be as small a single pipeline stage or consist of a collection of small cells that make up a larger re-usable library component. The reduced performance model of such cells can hide internal handshaking details and highlight the key performance parameters of the cells along with the critical interactions with its environment. The reduced performance models also greatly simplifies the combined performance model of any larger system comprised of these building blocks, making system-level performance analysis much more efficient. In this way, this work may form the basis to supplement any building-block based asynchronous design flow with efficient performance analysis.

In particular, we proposes two specific reductions, projection and redundancy removal, and proves that these reductions are in the identified class of reductions that preserve TSE statistics. This work demonstrates the utility of these reductions by reducing a detailed Petri net describing the four-phase protocol of a linear pipeline comprised of well-known pre-charged half-buffer (PCHB) pipeline buffers

[46]. The result is a much simpler model of the linear pipeline at more of an architectural level, highlighting the critical paths through the pipeline buffers.

1.4 Contributions of the Thesis

- We formalizes a new performance optimization area for deterministic (no choice) asynchronous circuits called pipeline optimization. *To the best of our knowledge, no automated tool exists to indicate the degree of pipelining (e.g., number of pipeline stages) needed to achieve a given performance.*

- Toward this goal, we proposes an abstract model of the circuit on which the basic pipeline optimization problem can be defined. This abstract model is sufficient to characterize a variety of pipelining schemes, including those from Williams and Caltech [84, 46]. Though this model abstracts controller design details and differences in completion sensing schemes, it can be used effectively for initial micro-architecture exploration.

- Given that the basic synchronous retiming problem can be optimally solved in polynomial time [44], we first explored the complexity of our optimization problem. One contribution of this research is a proof that defined asynchronous pipeline optimization problem is NP-complete.

- In addition, we present an efficient branch and bound algorithm which demonstrates the feasibility of the optimization problem for moderately-sized models.

- As a practical approach, we consider an important sub-problem of the general asynchronous pipeline optimization problem called *slack optimization.* In this sub-problem all functional unit pipelining is fixed and we must add pipeline buffers to optimize performance. We explored the complexity of slack optimization problem and proved that defined slack optimization problem is also NP-complete.

- In addition, we present an efficient branch and bound algorithm which demonstrates the feasibility of the optimization problem for moderately-sized models.

- We characterized a class of Petri net reduction that preserve the bounds on TSE statistics as computed using the algorithms in [88, 86].

- We propose two specific reductions, projection and redundancy removal, and proves that these reductions are in the identified class of reductions that preserve TSE statistics.

- An case study demonstrates the utility of these reductions by reducing a detailed Petri net describing the four-phase protocol of a linear pipeline comprised of well-known pre-charged half-buffer (PCHB) pipeline buffers [46].

The result is a much simpler model of the linear pipeline at more of an architectural level, highlighting the critical paths through the pipeline buffers.

1.5 Thesis Organization

The organization of the remainder of the thesis is as follows. The thesis has two major parts. The first part, Chapter 4 to 6, describes pipeline optimization for asynchronous circuits. The second part, Chapter 7 to 11, describes probabilistic timed Petri nets reduction for asynchronous architectural analysis.

Chapter 2 presents pipeline analysis background, Chapter 3 describes the model on which we formulate the optimization problem, Chapter 4 proves NP-completeness of our problem, and Chapter 5 describes a relatively efficient exact solution based on a branch and bound algorithm with some preliminary experimental results. Chapter 6 then presents our proposed steps to complete the thesis.

Chapter 7 reviews the bounding algorithm for TSE statistics. Chapter 8 and 9 introduce a class of reduction operators that preserves TSE bounds and two specific reduction algorithms within this class. Chapter 10 and 11 consist of our PCHB case study and some conclusions.

Chapter 2

Background

This section first introduces the basics of asynchronous circuit design and classifies many of the existing asynchronous circuits design styles according to data encoding method, handshaking style, granularity of pipelines, and circuit style. Then, we introduce Petri nets as our performance analysis and optimization model.

2.1 Asynchronous Circuit Design Styles

Asynchronous circuits generally consist of functional units that communicate control and data information via *channels*. Channel implementations vary widely across different asynchronous design methodologies and can be classified by how they represent data and the handshaking protocols used to send the data.

2.1.1 Channel implementations: data encoding

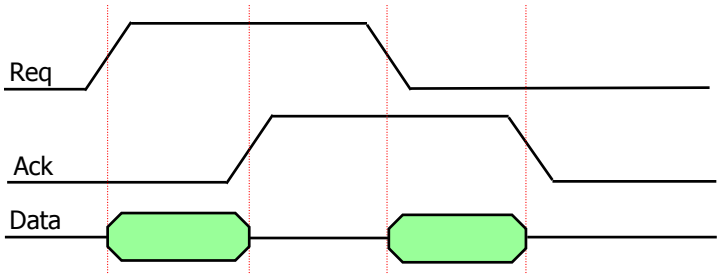
Single-rail channels [64] consist of one request wire and one wire per data bit from sender to receiver and one acknowledgment wire from receiver to sender. *Dual-rail channels* often consist of two wires per data bit from sender to receiver and one acknowledgment wire from receiver to sender. In addition, dual-rail designs can have an additional separate request line [68]. *1-of-N rail* channels are generalizations of dual-rail channels in which $\lceil log_2 N \rceil$ bits are sent using N wires. For instance, to send 3 bits, one might use three dual-rail channels or one 1-of-8 rail channel.

An acknowledgment signal from the receiver to the sender is used to tell the sender that the data is no longer needed. The logic that drives this acknowledgment signal often involves *completion sensing circuitry* that helps determine when the receiver is done using the current data bits. In single-rail channels, completion sensing circuits are implemented with *bundled data* lines [64] or more sophisticated *speculative completion sensing* circuitry [58, 7, 81] that includes delay lines that match the critical paths of the functional unit. On the other hand, completion sensing of dual-rail designs can be done using specialized logic that actively identifies when the computation is done. This latter logic relies on the dual-rail nature of the data and can be implemented without relying on timing assumptions and thus, is more robust to variations in delay than its delay-line counterparts. Completion sensing, however, requires more circuitry than delay lines and, if not done wisely, can incur a significant performance, power, and area penalty.

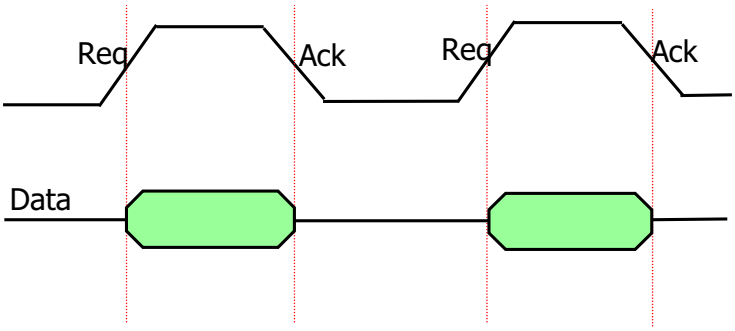
The functional units can be implemented using static or dynamic logic. Often functional units that communicate using dual-rail or 1-of-N channels are implemented using dual-rail dynamic logic [84, 46], but static logic is also possible [84]. Functional units that communicate using single-rail channels are more commonly implemented using static logic that is often smaller and consumes less power than dynamic counterparts. Designs implemented with dynamic logic, however, can generally achieve higher throughput than their static logic counterparts. Consequently, they can run at lower voltages to achieve a given throughput requirement and, thus may yield a lower power design than their their single-rail counterparts.

2.1.2 Channel Implementations: Handshaking styles

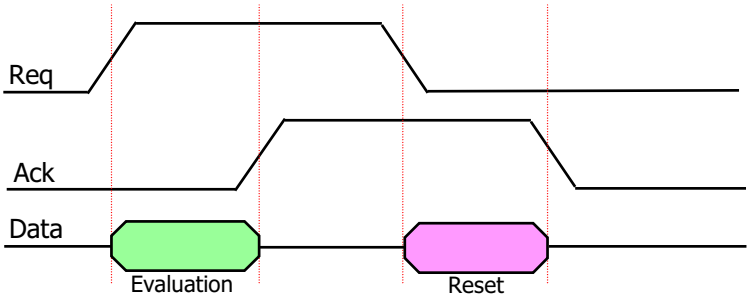
The various handshaking protocols for communicating across channels can be divided into *four-phase handshaking* [29] and *two-phase handshaking* [77]. In the two-phase handshake protocol, a request and an acknowledge wire are used to implement handshaking between the sender and receiver. The typical operation of two-phase handshaking is depicted in Figure 2.1(a). In the two-phase handshake protocol, all transitions are functional and consequently every pair of consecutive request/acknowledge transitions forms a complete handshake. *Two-phase single-rail* [27] channels are usually seen with static logic functional units that use bundled-data for completion sensing. Due to some difficulties in designing complex two-phase control circuits, a novel *single-track handshaking protocol* has been suggested



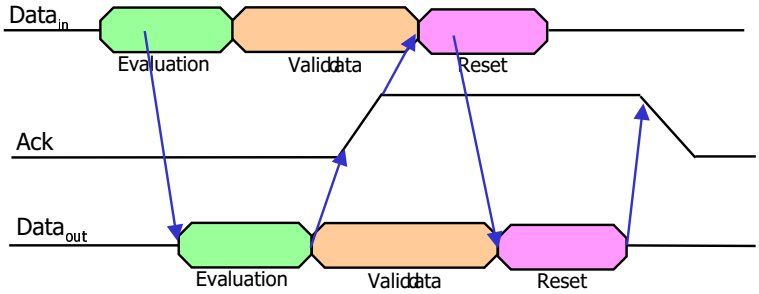
(a) Two-phase handshaking protocol



(b) Single track handshaking protocol



(c) Four-phase handshaking protocol



(d) Four-phase handshaking protocol

Figure 2.1: Two-phase handshaking protocol vs. four-phase handshaking protocol

by Van Berkel and Bink [9]. This handshaking protocol is achieved by combining the request and acknowledge line onto one wire and is illustrated in Figure 2.1(b). Sutherland et al. later developed faster single-rail GasP pipeline to control fine-grain bundled data pipelines [76]. Recently, single-track fine-grain pipeline templates without bundling constraints are developed by Ferretti et al. [26] and Nyström et al. [59]. Though two-phase handshaking and its modified single-track version are used in single-rail designs, two-phase dual-rail design is rarely used. This may be because it difficult to find time to precharge the functional units in the two-phase protocol.

Where two-phase handshaking involves two events per cycle, four-phase handshaking requires four events, as shown in Figure 2.1(c). Since four events are used to designate a complete handshaking cycle, half of these are essential for functional computation and the other half are not actively used to communicate data. Nevertheless, this reset phase is very useful for precharging dynamic functional units. Figure 2.1(d) shows a four-phase handshaking protocol for dual-rail dynamic units [84, 46]. Other protocols extend the data valid region through the reset phase [64, 13] to more efficiently use four-phase handshaking with static functional units.

2.1.3 Delay models

Most design techniques require some timing assumptions or constraints on the wires and/or components to ensure correct operation. For example, in synchronous circuit

design, the data input to every register must satisfy all setup and hold times. The delay assumptions in asynchronous circuits widely vary based on design styles as outlined below.

- *Delay insensitive (DI)*: DI designs [82, 24] require no timing assumptions on either wires or gates. That is, DI circuits work correctly for any arbitrary, time-varying gate and wire delays. This is the most conservative and robust approach asynchronous design style, but it has been shown that very few gate-level DI designs can exist [48].

- *Quasi-delay insensitive (QDI)*: QDI design [47, 46] is a practical approximation to DI design. QDI circuits work correctly regardless of delays in gates and all wires except in cases of wire forks designated *isochronic*. The difference in time at which the signal arrives at the ends of an isochronic fork must be less than the minimum gate delay. If these isochronic forks are guaranteed to be local to a small component, these circuits can be practically as robust as DI circuits. The QDI assumption has also be extended to include assumptions of isochronic propagation through a number of logic gates (extended isochronic fork and Q^nDI delay models [10]).

- *Speed independent (SI)*: SI design [84, 5] assumes that gate delay can be arbitrary but that all wire delay is negligible. From a delay perspective SI designs basically assume that all forks are isochronic. For the design of small control circuits, this timing assumption is generally satisfied.

- *Scalable delay-insensitive (SDI)*: SDI approaches [57, 56] are motivated by the observation that SI design should not be used for any circuit that spans significant chip area. Consequently, in SDI design the chip area is divided into many regions, SI circuit design is used within each region, and communication between regions is done with DI channels.

- *Bounded delay:* In bounded delay models each gate is given a minimum and maximum delay and the circuit must work if the delay of all gates are within these bounds. These *timed circuits* can often be faster, smaller, and lower power than their QDI or SI counterparts but require more careful timing verification during physical design [54].

- *Relative-timing:* In relative-timing-based circuits, a list of path constraints identifies sets of path pairs, where for each pair of paths, one path must be must be longer/shorter than each other to ensure correctness. These circuits can have the same benefits of timed circuits and may be easier to validate [75, 42, 19].

2.2 Stochastic Timed Petri nets

Petri nets [53] are a graphical means of specifying and modeling concurrency, synchronization, and choice. They are widely used in the fields of system analysis, asynchronous circuit synthesis, and formal verification. Some specialized Petri nets

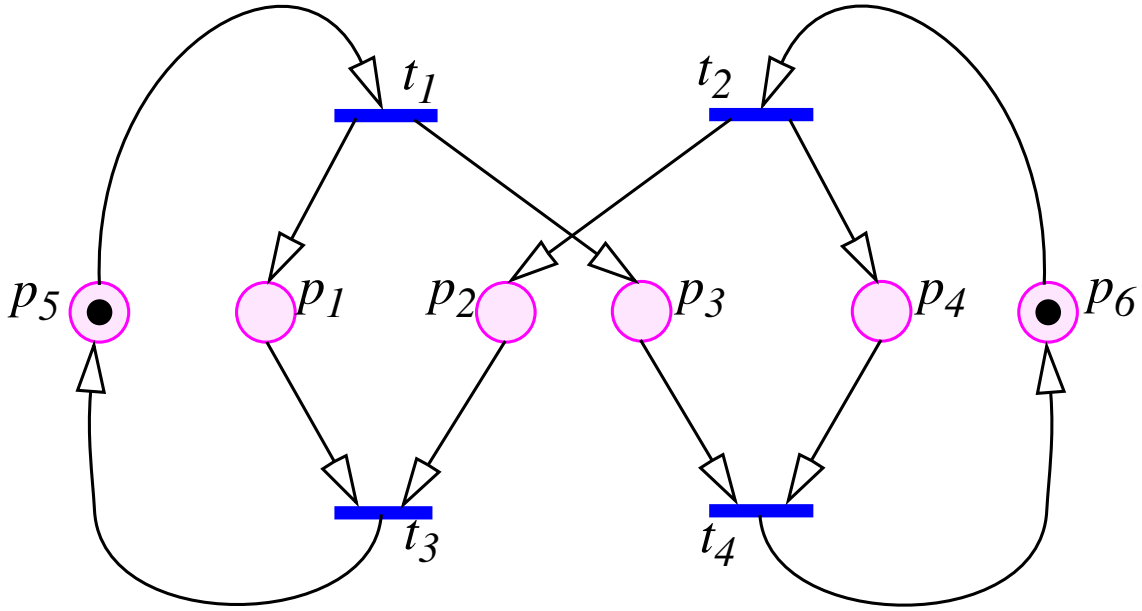


Figure 2.2: An illustrating example of a Petri net.

can be annotated with timing and delay information making them well-suited for performance analysis in a wide range of applications [15, 35, 52, 53], including asynchronous circuits [84, 88]. This section reviews general concepts of (timed) Petri net. More detailed review of Petri nets can be found in a survey paper [53].

2.2.1 Petri net models

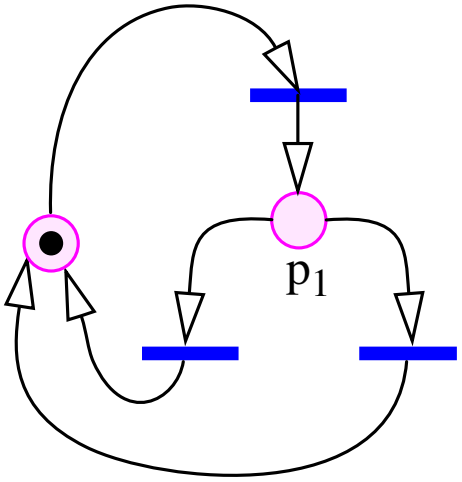
A Petri net is a triple $N = (P, T, F)$ where P is the finite set of places, T the finite set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ the flow relation. We say for an element $x \in (P \cup T)$, that $\bullet x$ is the preset of x defined as $\bullet x = \{y \in P \cup T | (y, x) \in F\}$ and $x\bullet$ is the postset of x defined as $x\bullet = \{y \in P \cup T | (x, y) \in F\}$.

Figure 2.2 shows an example of a Petri net where $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}, T = \{t_1, t_2, t_3, t_4\}$ and $F = \{(t_1, p_1), (t_1, p_3), (p_1, t_3)...\}$.

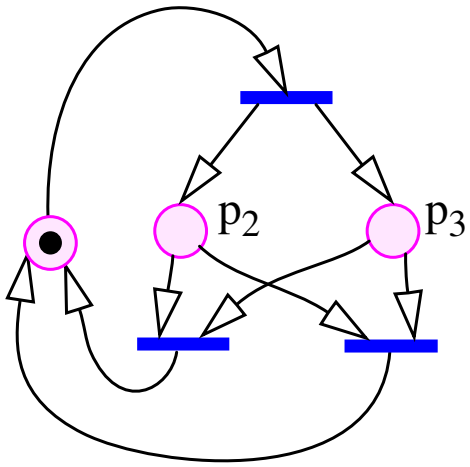
A Petri net is called a *state machine* if every transition has at most one input and one output place, i.e., $|\bullet t| \leq 1 \wedge |t\bullet| \leq 1, \forall t \in T$. *A marked graph* [18] is a type of Petri net in which every place has at most one input and output transition, , i.e., $|\bullet p| \leq 1 \wedge |p\bullet| \leq 1, \forall p \in P$. A state machine can model *choice* but not *concurrency* whereas marked graph can model *concurrency* but not *choice*.

When a place p has more than one postset transition, that is $|p\bullet| > 1$, we say p is a choice place. There are a few sub-types of choice places. Place p is an extended free choice place if all transitions in $p\bullet$ can be enabled regardless of external influence, in order words, selection of the transition to fire is non-deterministic. A special case of extended-free choice is a free-choice place p for which all output transitions of p have only one input place p. A place p is a unique choice place, if no two output transitions of p can be enabled at the same time. In this case, the selection of which transition to fire is deterministic. Any choice place which is neither extended free choice nor unique choice is classified as arbitration choice. The mathematical definitions and graphical examples of the various kinds of choices are shown in Figure 2.3.

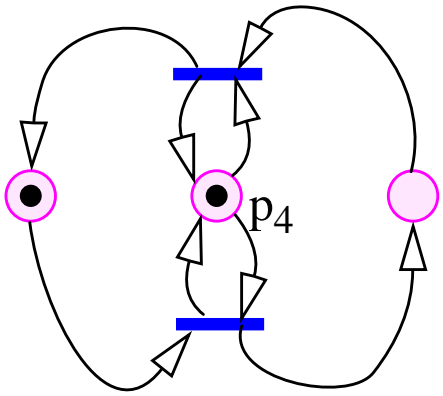
A marking is a token assignment for the place and it is represents the state of the system. Formally, a marking is a mapping $M : P \rightarrow \{0, 1, 2, \cdots\}$ where the



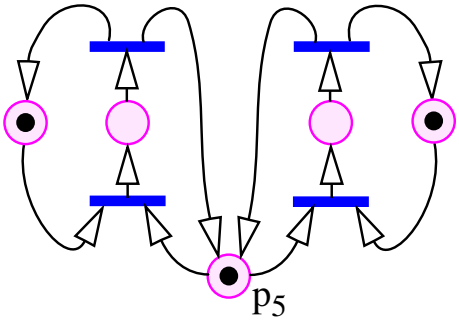
(a) A Petri net with free choice (P_1): $\bullet(p\bullet) = \{p\}$



(b) A Petri net with extended free choice (p_2, p_3): $\forall p' \in P : (p\bullet \cap p'\bullet = \emptyset) \vee (p\bullet = p'\bullet)$



(c) A Petri net with unique choice (P_4)



(d) A Petri net with arbitration choice (P_5)

Figure 2.3: Petri nets with choice.

number of tokens in place p under marking M is denoted by $M(p)$. If $M(p) > 0$, then the place p is in the support set of the net.

The firing rule of Petri nets is defined as follows. A transition t is *enabled* at marking M if $M(p) \geq 1, \forall p \in {}^\bullet t$. An enabled transition may fire. The firing of t removes one token from each place in its preset and deposits one token to each place in its poset, leading to a new marking M', denoted by $M[t\rangle M'$. A sequence of transitions $\sigma = t_0 t_1 \cdots t_{m-1}$ is a firing sequence from a marking M_0 iff $M_k[t_k\rangle M_{k+1}$ for $k = 1, \cdots, m-1$. In this case, we write $M_0[\sigma\rangle M_m$ and say σ has a length of m denoted by $|\sigma|$. We write $\vec{\sigma}$ to be the firing counter vector of σ, indicating the numbers of times transitions are fired along σ. The vector is called a *T-invariance* if there is a marking M such that $M[\sigma\rangle M$. The transition firing rule itself represents a type of max constraint in the sense that *all* preset places of a transition must be marked to enable a transition to fire.

A *marked* net Σ is a tuple (N, M_0), where N is a net and M_0 is its initial marking. The set of reachable markings of Σ is denoted by $R(M_0)$. Σ is *live* iff all transitions will eventually be enabled from every $M \in R(M_0)$. It is *k-bounded* if $M(p) \leq k$ for $\forall M \in R(M_0), \forall p \in P$. A 1-bounded net is also called *safe*. A live and bounded (LB) marked net has no source or sink places and no source or sink transitions (e.g.,[53]). Thus, a LB net can be partitioned into a set of strongly connected components each evolving independently of others. Below, we assume

the net is strongly connected. In particular, we restrict ourselves to LB nets with free-choice and unique-choice places.

2.2.2 Timing and choice probabilities

In some variants of Petri nets, each transition or place can be annotated with delay information such as a fixed delay, a pair of values denoting lower and upper bounds of delay or even more general stochastic delay distributions [88, 70]. These Petri nets with timing constraints are often called *timed Petri nets* [83] and are further classified as *timed place Petri nets (TPPN)* and *timed transition Petri nets (TTPN)* depending on the whether the timing bounds annotate places or transitions.

In this work, we associate delay with places. That is, a token flowing into a place p experiences a delay associated with p before it can be consumed by an output transition of p. The delays experienced with different places are independent. The actual firing of a transition is assumed to be *instantaneous*. For a place p, we write $X(p)$ as the random variable denoting the delay associated with it, and write $F_{X(p)} : R \to [0, 1]$ as the distribution function of $X(p)$, i.e., $F_{X(p)}(x) = \text{Prob}(X(p) \leq x)$. Unlike some other research, we do not put any restriction on the distribution functions except that they all have finite moments up to a sufficiently high order.

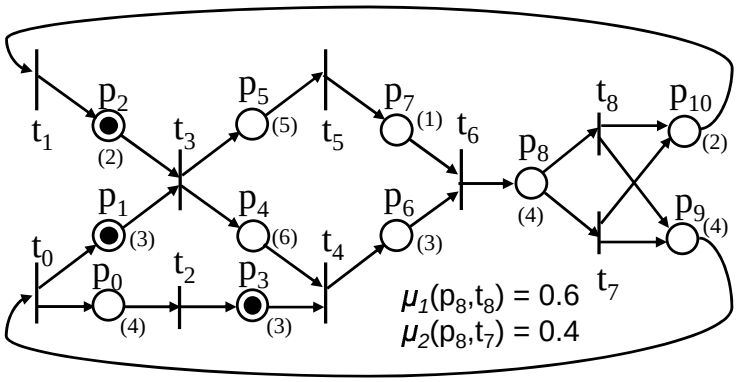
For Petri nets with free- or unique- choices, these assumptions imply there is no race condition among transitions in structural conflict, i.e., the poset of a choice place. For each choice place p, we assume there is a probability mass function

(p.m.f) $\mu(p, \cdot)$ to resolve the choice. That is, if $t \in p\bullet$, $\mu(p, t)$ is the probability that t consumes the token each time p is marked. Of course, $\sum_{t \in p\bullet} \mu(p, t) = 1$. P_c, P_{nc} denote a set of all choice places, and a set of all places without choice respectively.

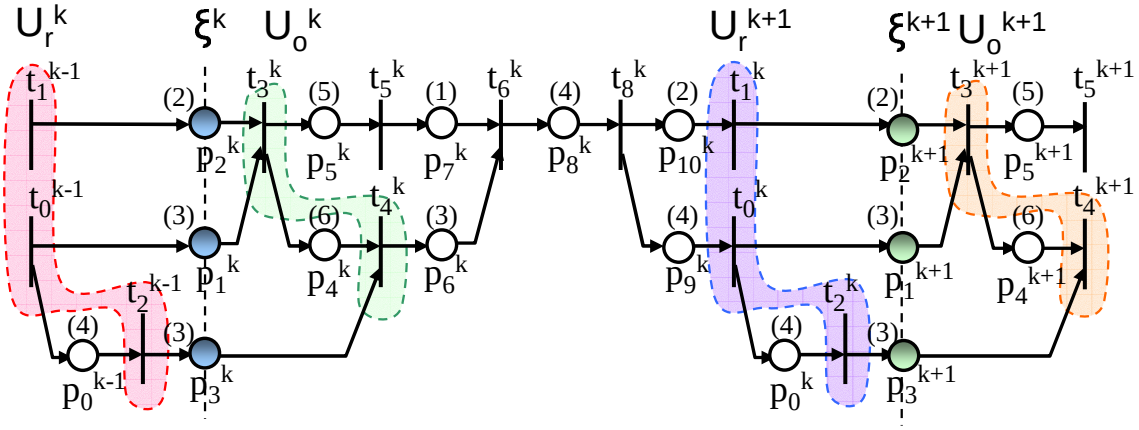
We call these Petri nets, *Stochastic timed Petri Nets (STPN)* [88] and the subset of STPN which allow only fixed delays, *probabilistic timed Petri nets (PTPN)* [43]. Figure 2.4(a) shows an example of such a net where P_8 is free-choice place with probability mass function.

We say ψ is a deterministic path in Petri net N if it is a sequence of non-choice places connected by transitions. The set of all deterministic paths leading from x to y is denoted by $\Psi(x, y)$ where $x, y \in P_{nc} \cup T$. Let $\Delta(\psi)$ be sum of delay along that deterministic path, that is, $\Delta(\psi) = \Sigma_{p \in \psi} d(p)$. Let $M(\psi)$ be sum of tokens along that deterministic path, that is, $M(\psi) = \Sigma_{p \in \psi} M(p)$.

For the purpose of performance analysis and architectural modeling of the asynchronous system, we adopt STPN and PTPN in which we annotate delay for each place and a probability mass function f to each free choice place. In fact, for the first part of this work, we will limit ourselves to choice-free nets (i.e., marked graphs) which are sufficient to model and analyze the performance of deterministic asynchronous pipelines. Specifically, each cycle in the marked graph is associated with a *cycle metric* that is the sum of the delays of all associated transitions along the cycle divided by the maximum number of tokens that reside in the cycle in any reachable marking of the graph. The *cycle time* of a deterministic pipeline is



(a) A PTPN



(b) A timed execution of the PTPN in (a)

Figure 2.4: A probabilistic timed Petri net and its timed execution.

defined as the largest cycle metric in its marked graph representation [14, 65]. This cycle time is analogous to the cycle time of a synchronous circuit and identifies the throughput of the design.

2.2.3 Timed executions

We call a possible run of an STPN a *timed execution* where choices are resolved and places are assigned delay values. In particular, we call a firing of a transition an *event*. A timed execution can be described as a sequence of events and their occurrence times. Alternatively, it can be depicted as a timed event graph which describes the causality among events.

For example, Figure 2.4(b) shows the event graph of a timed execution of the PTPN in Figure 2.4(a). The numbers along the (instanced) places denote the delay values. For convenience, we write t^k and p^j to denote the k-th event due to the firing of t and the j-th instance of place p, respectively.

Formally, a timed execution π of Σ is a triple (N_π, d_π, ℓ) where $N_\pi = (P_\pi, T_\pi, F_\pi)$ is an acyclic event graph, d is a function $P_\pi \rightarrow \mathbf{R}$ that denotes the delay value of each place in P_π and a labeling function $\ell : P_\pi \cup T_\pi \rightarrow P \cup T$ that maps each places and transitions of N_π to their corresponding ones in Σ. We use a function τ (called timing function) to denote the occurrence times of events. For the timed execution shown in Fig. 2.4(b), the corresponding functions d and τ are illustrated along the

places and transitions. For a given timed execution, the occurrence time of event $t^{(k)}$ is determined as follows:

$$\tau(t^{(k)}) = \max_{\substack{(s^{(j)},p) \in F_\pi, \\ p \in \bullet t^{(k)}}} \tau(s^{(j)}) + d(p) \tag{2.1}$$

where the term $\tau(s^{(j)}) + d(p)$ reduces to $d(p)$ if p is a source place of π.

Note that for STPNs with only free- and unique-choice, the set of event graphs resulting from all possible timed executions coincides with the set of all possible untimed processes of the net (e.g.,[38]). This fact will be exploited later to decouple the choice resolutions from the timing behavior.

2.2.4 TSEs and their statistics

Given a timed execution π, the Time Separation of an Event pair (TSE) is the time distance between the two events. More precisely, the TSE of event pair $(s^{(k)}, t^{(k+\varepsilon)})$, denoted by, $\gamma^{(k)}(s,t,\varepsilon)$, is:

$$\gamma^{(k)}(s,t,\varepsilon) = \tau(t^{(k+\varepsilon)}) - \tau(s^{(k)}) \tag{2.2}$$

where τ is the timing function of π, and (s,t,ε) is called the corresponding *separation triple*. When π is viewed as a random timed execution, the TSE $\gamma^{(k)}(s,t,\varepsilon)$

is a random variable, and the sequence $\{\gamma^{(k)}(s,t,\varepsilon) : k = 1, 2, \cdots\}$ is a random process.

Definition 1 *The average TSE due to separation triple* (s,t,ε)*, denoted by* $\overline{\gamma}(s,t,\varepsilon)$*, is the average of the corresponding TSEs of an infinite timed execution of the STPN. That is,*

$$\overline{\gamma}(s,t,\varepsilon) = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \gamma^{(k)}(s,t,\varepsilon). \tag{2.3}$$

As we will show later, many system performance metrics such as average throughput and latency can be directly expressed as the average TSEs of some indicating event pairs.

Definition 2 *The variance of the TSEs due to separation triple* (s,t,ε) *is:*

$$\sigma^2_{\gamma(s,t,\varepsilon)} = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} |\gamma^{(k)}(s,t,\varepsilon) - \overline{\gamma}(s,t,\varepsilon)|^2 \tag{2.4}$$

Equivalently, $\sigma^2_{\gamma(s,t,\varepsilon)} = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} {\gamma^{(k)}}^2(s,t,\varepsilon) - \overline{\gamma}^2(s,t,\varepsilon)$.

Definition 3 *The frequency function of the TSEs due to separation triple* (s,t,ε) *is a mapping* $F_{\gamma(s,t,\varepsilon)} : \mathbf{R} \to [0,1]$ *such that*

$$F_{\gamma(s,t,\varepsilon)}(x) = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \mathbf{1}_{\gamma^{(k)}(s,t,\varepsilon)\le x} \tag{2.5}$$

where the indicating function $\mathbf{1}_A$ *evaluates to 1 if the subscript expression A is true, and 0 otherwise.*

To characterize the above defined statistics, it is natural to first question their existence. There are cases where an average TSE does not exist. The difference between the numbers of occurrences of two transitions might be infinite almost surely as time progresses. Consequently, their average TSE diverges almost surely for any finite occurrence offset ε provided that the net has at least one place with a positive mean delay. [86]

However, for a wide class of TSE pairs, their average TSEs exist [88]. Condition 1 below formally characterizes such a class with the aid of the notion of *steady markings*. A steady marking[1] is one that can be reached from all reachable markings.

Condition 1 *Let t and s be two transitions of Σ. If M is a steady marking and σ is a firing sequence such that $M[\sigma\rangle M$, then $\vec{\sigma}(s) = \vec{\sigma}(t)$.*

The condition requires that the net fires both transitions the same amount of times in order to traverse any cycle of steady markings. In practice, we expect this condition to be guaranteed by the user. However, it can also be checked using structural analysis for free-choice nets and a reachability analysis (e.g., [85]) for nets with both free-choice and unique-choice.

The following theorem [88] verifies that when transitions (s, t) satisfies Condition 1, their TSE sequence generated by a random timed execution is *weakly ergodic* and thus its average exists.

[1]All the steady markings of a LB Petri net with unique- and free-choices make up the unique strongly connected component of the reachability graph of the net [12].

Theorem 1 *Let Σ be a LB Petri net that has only free-choices and unique-choices and satisfies stochastic assumptions given in Section 2.3. For any transition pair (s,t) for which Condition 1 holds, its corresponding average TSE with a fixed occurrent offset ε is a finite constant $\overline{\gamma}(s,t,\varepsilon)$ almost surely and in mean. That is,*

$$\text{Prob}(\lim_{n\to\infty}\frac{1}{n}\sum_{k=1}^{n}\gamma^{(k)}(s,t,\varepsilon)=\overline{\gamma}(s,t,\varepsilon))=1, \tag{2.6}$$

$$\lim_{n\to\infty}\frac{1}{n}\sum_{k=1}^{n}\mathbf{E}\gamma^{(k)}(s,t,\varepsilon)=\overline{\gamma}(s,t,\varepsilon). \tag{2.7}$$

In some cases, the average time distance between some special occurrences of two transitions still exists although the two transitions do not satisfy Condition 1.

Case 1 *The interested average time distance is between an occurrence of s to the ε-th occurrence of t following the occurrence of s.*

Such an average time distance may make sense if s occurs less frequently than t, although it does not satisfy the definition of average TSE set earlier. If one can skip the irrelevant occurrence of t, Condition 1 is met.

Case 2 *The pair (s,t) is such that for every T-invariant $\vec{v}$, if $\vec{v}(s)>0$, $\vec{v}(t)>0$, and vice versa. Moreover, $\vec{v}(s)/\vec{v}(t)$ is a constant.*

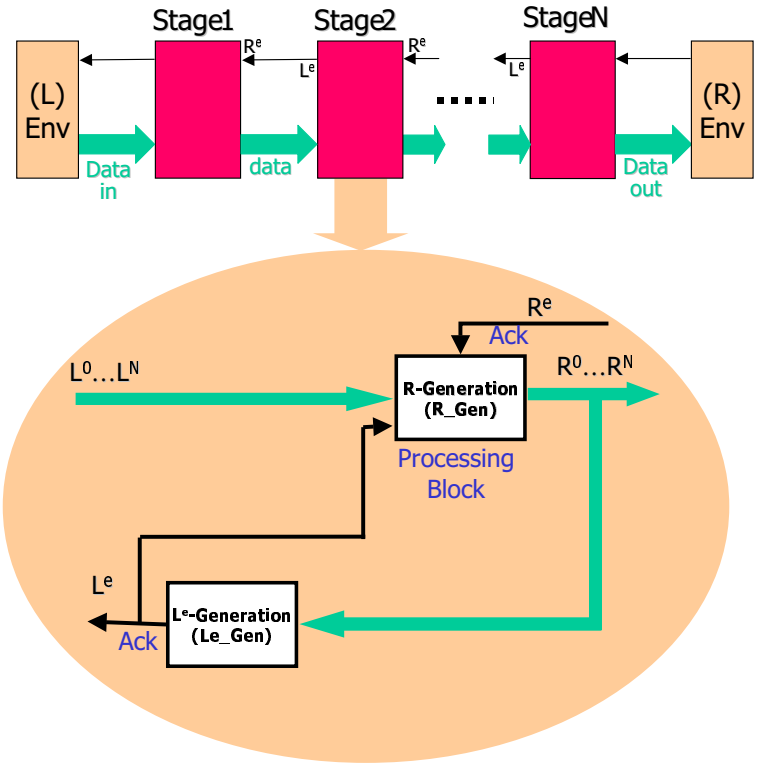
In this case, one may split the occurrences of each of the two transitions into the occurrences of some new transitions, and consider the average TSE of a pair of new transitions corresponding to s and t. The new transition pair satisfies Condition 1.

Below, we assume the interested transition pair satisfies Condition 1 which serves as the basis of the other two cases.

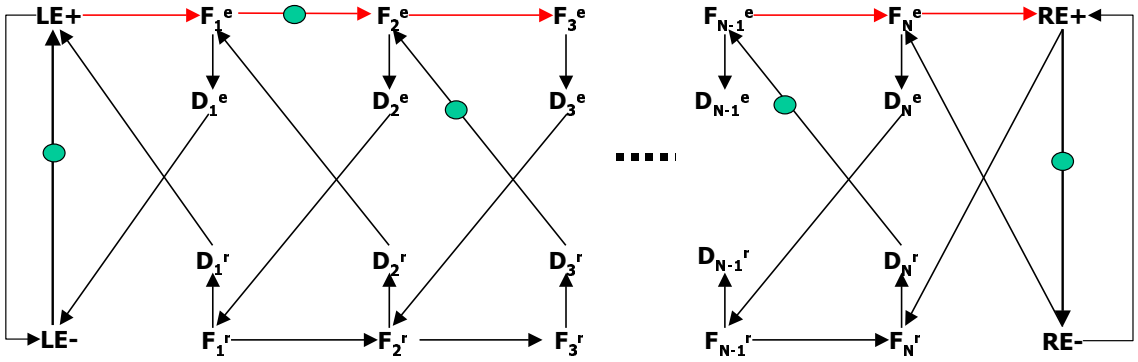
The existence of TSE variance in the STPNs requires more consideration. In practice, the delays on places are finite or their moments are finite up to a sufficiently high order. In such cases, when a TSE sequence is weakly ergodic (e.g., as characterized by Theorem 1), the existence of its variance (and in fact, any of its finite-order moments) is guaranteed [86]. Similarly, when the TSE sequence is weakly ergodic, its frequency function defined as the limit in (2.5) exists at every fixed point almost surely and in mean. In that case, we also call the frequency function as the *distribution* of the TSE. [86]

2.3 Asynchronous Pipelines and Their Performance Models

As mentioned earlier, numerous different asynchronous pipeline templates have been proposed by various research and industrial teams [46, 84, 80, 2, 72, 71, 76, 60, 26]. In this section, we describe a few of these schemes along with their proposed marked graph performance model.

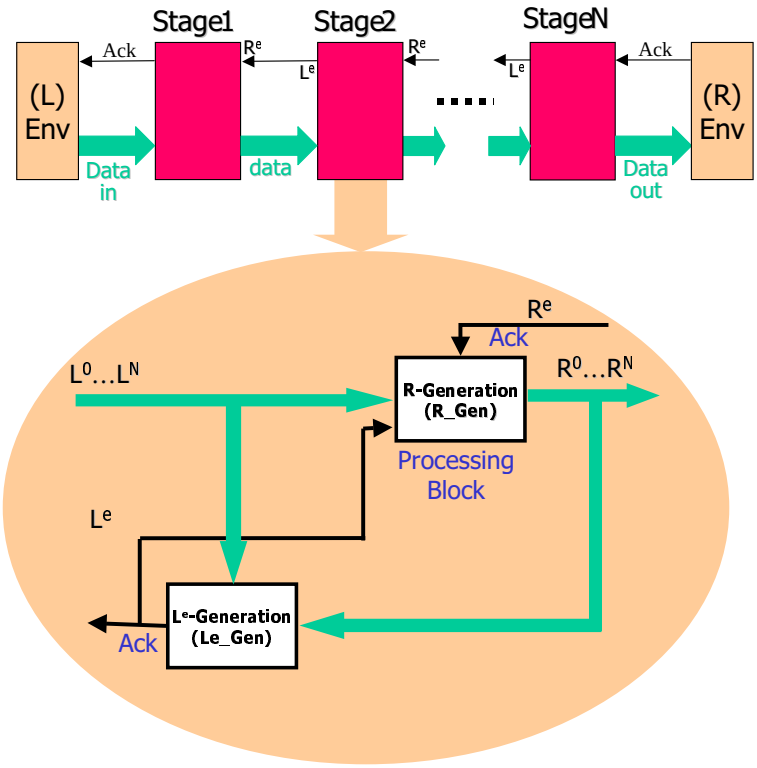


(a) WCHB pipeline scheme.

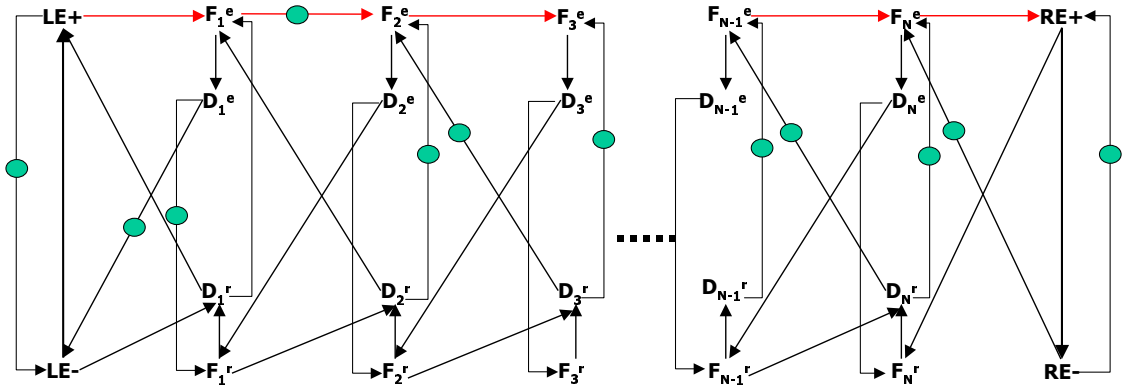


(b) WCHB marked graph model.

Figure 2.5: Caltech's WCHB pipeline: Note that L^e and R^e are inverted values of L^a and R^a, often used for convenience.



(a) PCHB pipeline scheme.



(b) PCHB marked graph model.

Figure 2.6: Caltech's PCHB pipeline: Note that L^e and R^e are inverted values of L^a and R^a, often used for convenience.

2.3.1 Four Phase QDI Pipelines

Caltech researchers model asynchronous circuits using Communicating Sequential Processes (CSP) [34]. The basic asynchronous pipeline stage, which has a left channel L on which it accepts inputs and a right channel R on which it communicates outputs, has the CSP specification *[L;R]. One specific form, which has a passive handshaking protocol for L and a lazy active protocol for R, has the following four-phase handshaking protocol.

$$*[[L]; \quad L^a \uparrow; \quad [\neg L]; \quad L^a \downarrow; \quad [\neg R^a]; \quad R \uparrow; \quad [R^a]; \quad R \downarrow].$$

$[L]$ indicates a wait for the validity of the input data on channel L while $[\neg L]$ indicates a wait for the neutrality the L input data. $[R^a]$ indicates a wait for the acknowledgment that the data on the output channel R has been received, and $[\neg R^a]$ indicates a wait for the reset of this acknowledgment. $L^a \uparrow$ indicates the raising of the input acknowledgment wire (indicating that the input data has been received), and $L^a \downarrow$ indicates lowering the input acknowledgment wire. $R \uparrow$ means that all the outputs are set to their valid states and $R \downarrow$ means that all the outputs are reset to their neutral states. This particular four-phase handshaking protocol is not very concurrent. The resetting of both the L and R channels are in the critical path and thus represent pure overhead. In his Masters thesis [46], Lines investigated all possible event reshufflings that can improve performance. He argues that among

the many possible reshufflings, only a few are typically useful. In particular, two that he identifies as particularly useful are the weak condition half buffer (WCHB) and the precharge half buffer (PCHB). These pipeline schemes, depicted in Figures 2.5(a) and 2.6(a), are quasi-delay insensitive and thus are very robust to delay variations. The detailed reshufflings are:

$$WCHB \quad \equiv \quad *[[\neg R^a \wedge L]; \quad R\uparrow; \quad L^a\uparrow; \quad [R^a \wedge \neg L]; \quad R\downarrow; \quad L^a\downarrow;]$$

$$PCHB \quad \equiv \quad *[[\neg R^a \wedge L]; \quad R\uparrow; \quad L^a\uparrow; \quad [R^a]; \quad R\downarrow; \quad [\neg L]; \quad L^a\downarrow;]$$

To analyze the performance (i.e., throughput) of these protocols, we present their marked graph representations, determine all cycles, and calculate their largest cycle metric. The marked graph for a N-stage pipeline in which each pipeline stage uses the WCHB protocol is depicted in Figure 2.5(b).[2] To analyze the performance of these pipelines, we introduce delay information regarding each transition. In particular, the i^{th} pipeline stage is associated with a *function evaluation delay* $\tau(F_i^e)$, a *function reset delay* $\tau(F_i^r)$, a *completion sensing delay for evaluation* $\tau(D_i^e)$, a *completion sensing delay for reset* $\tau(D_i^r)$, a *control overhead delay for evaluation* $\tau(C_i^e)$, and a *control overhead delay for reset* $\tau(C_i^r)$. In this marked graph, there

[2]Note that the places in this and all later marked graph models are omitted for brevity.

exists three cycles, each containing only one-token, for every sequence of three pipeline stages. The cycle metric for each of these cycles is as follows:

$$\tau(F_i^e) + \tau(F_{i+1}^e) + \tau(F_{i+2}^e) + \tau(D_{i+2}^e) + \tau(F_{i+1}^r) + \tau(D_{i+1}^r),$$

$$\tau(F_i^r) + \tau(F_{i+1}^r) + \tau(F_{i+2}^r) + \tau(D_{i+2}^r) + \tau(F_{i+1}^e) + \tau(D_{i+1}^e),$$

$$\tau(F_i^e) + \tau(F_{i+1}^e) + \tau(D_{i+1}^e) + \tau(F_i^r) + \tau(F_{i+1}^r) + \tau(D_{i+1}^r)$$

Other cycles may exist due to algorithmic dependencies that arise in non-linear pipelines (loops, fork-join structures, etc.). As mentioned earlier, the maximum cycle metric determines the average throughput of the pipeline.

For the PCHB marked graph model, depicted in Figure 2.6(b), a sequence of three stages yields three different cycles with the following cycle metrics:

$$\tau(F_i^e) + \tau(F_{i+1}^e) + \tau(F_{i+2}^e) + \tau(D_{i+2}^e) + \tau(F_{i+1}^r) + \tau(D_{i+1}^r),$$

$$\tau(F_i^e) + \tau(F_{i+1}^e) + \tau(D_{i+1}^e) + \tau(F_i^r) + \tau(D_{i+1}^r),$$

$$\tau(F_i^e) + \tau(D_i^e) + \tau(F_i^r) + \tau(D_i^r)$$

WCHB is slightly less concurrent than PCHB, but has simpler (and faster) circuit realizations for some functions. Consequently, for different functions/pipelines, the preferred protocol may be different [46].

Recently, USC researchers developed two new pipeline templates, reduced stack precharge half buffer (RSPCHB) and reduced stack precharge full buffer (RSPCFB),

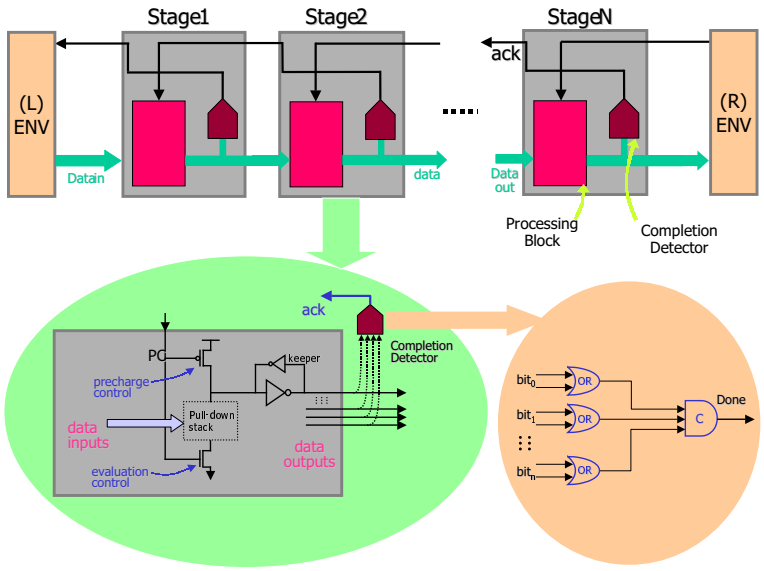
that provide performance improvement over Caltech's pipelines without sacrificing quasi delay insensitivity [61]. The key idea of these templates is to reduce the complexity of internal circuitry by reducing concurrency and using additional handshaking wire between stages.

2.3.2 Four Phase Pipelines with Timing Assumption

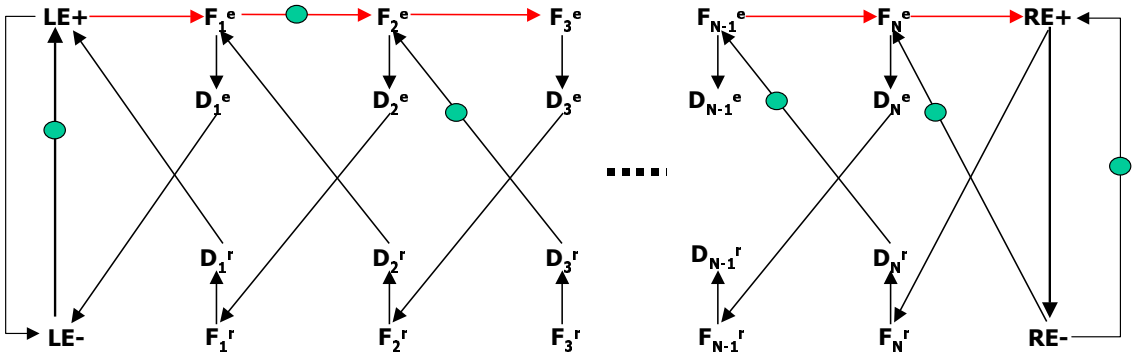
Figure 2.7 shows the basic PS0 pipeline protocol developed by Williams along with its marked graph representation [84]. Williams' PS0 has a faster cycle time than Caltech's pipeline template and PS0 circuits often are simpler, yielding faster and lower power circuit realizations. However, PS0 pipeline template has timing assumptions to operate correctly. Especially, general PS0 pipelines that contain forks and joins has global timing assumptions besides a local timing assumption [62]. To avoid these global timing assumption, couple of techniques are developed with costs of complicated additional control circuitry and/or weak-conditioned logic [62]. The other disadvantage of PS0 pipeline template is that it has the following local timing assumption

$$\tau(D^e_{i+1}) + \tau(F^r_i) > \tau(F^e_{i+2}) + \tau(D^e_{i+2}) + \tau(F^r_{i+1}) + \tau(D^r_{i+1})$$

which must be verified during physical design. (Note that this means that PS0 circuits are thus neither quasi-delay insensitive nor speed-independent.) For this



(a) PS0 pipeline scheme.



(b) PS0 marked graph model.

Figure 2.7: Williams' PS0 pipeline.

marked graph, a sequence of three stages of a linear pipeline yields the following one-token cycle:

$$\tau(F_i^e) + \tau(F_{i+1}^e) + \tau(F_{i+2}^e) + \tau(D_{i+2}^e) + \tau(F_{i+1}^r) + \tau(D_{i+1}^r)$$

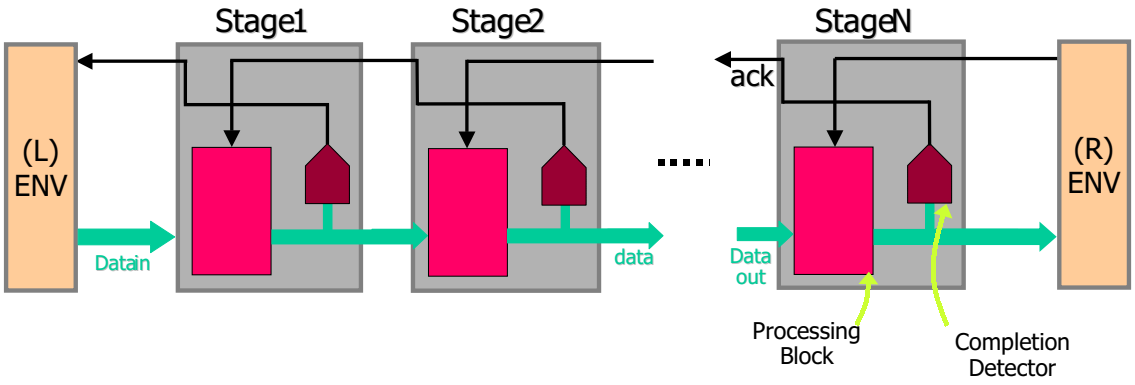
For non-linear PS0 pipelines that contain forks and joins, the above equations must be modified to include some control circuit overhead.

2.3.3 Null Conventional Logic - NCL

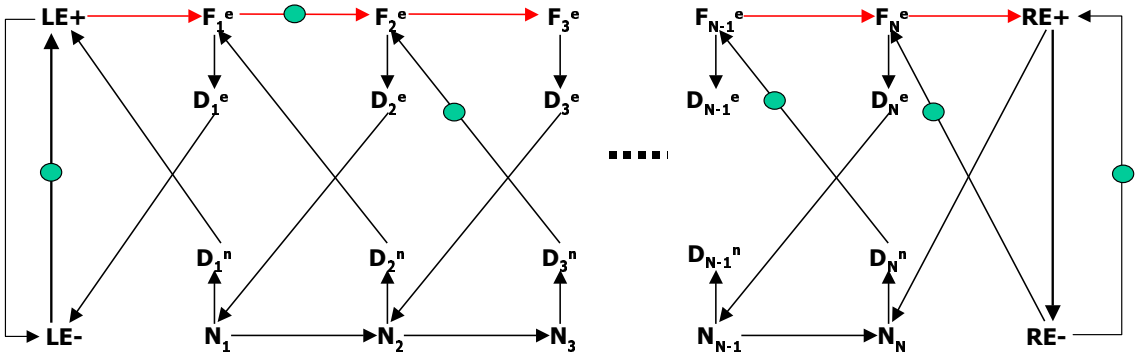
Null conventional logic, originally developed by Fant [25], has been commercially extended by Theseus Logic, Inc [80]. Pipeline control of NCL logic is similar to that of Williams' PS0. After stage i performs its computation, stage $i+1$ can compute, followed by stage $i+2$. Once stage $i+2$ performs its computation, the results from stage $i+1$ are no longer needed and their values can be reset. Afterwards, stage i can re-evaluate.

The marked graph model for a NCL pipeline is depicted in 2.8(b). Every three pipeline stages has one one-token cycle with the following cycle metric

$$\tau(F_i^e) + \tau(F_{i+1}^e) + \tau(F_{i+2}^e) + \tau(D_{i+2}^e) + \tau(N_{i+1}) + \tau(D_{i+1}^n)$$



(a) NCL pipeline scheme.



(b) NCL marked graph model.

Figure 2.8: NCL pipeline.

This cycle metric is similar to that of Williams' PS0 pipelines. Moreover, when applied to very fine grain pipelines, the timing assumptions required by both protocols are also similar. NCL logic, however, can be more complex than standard dual-rail logic and consequently, it may produce worse performance and/or power than Williams dual-rail PS0 designs. One key benefit of NCL logic, however, is a novel synthesis methodology that is able to use static logic and existing synchronous synthesis tools [45].

2.3.4 Single-track Pipelines

Single-track handshake circuits was introduced by Berkel et al. to control medium grain bundled-data [9]. As shown in Figure **??**, the sender waits for the wire to be low ("ready") before sending a request by driving the wire high ("busy"). After the receiver detects the wire is high and consumes the data, it drive the wire low.

Later, Sutherland et. al developed faster single-rail GasP circuits to control high performance fine-grain bundle-data pipeline [76, 22]. Figure **??** shows the GasP circuits. Initially, L, R, and A are high. When L is driven low by the left environment, the self-resetting Nand gate will fire, driving internal signal A low. This will restore L, active the data latches, and drive R low, propagating the signal and avoiding re-evaluation until R is restored high by the right environment. The self resetting Nand gate will restore itself by driving A high after 3 transactions. The output of the Nand gate controls the latches in a parallel single-rail datapath.

Recently, USC researchers developed Single Track Full Buffers (STFB) [26]. STFB templates uses 1-of-N data-encoding and two-dimensional pipelining instead of single-rail pipelining used by GasP. Figure **??** illustrates dual-rail STFB pipeline template as well as its marked graph performance model. Detail operation of the STFB is as follows. After some of the inputs are driven high by left environment, the n-stack gate will drive (S_i) low, thereby driving both the corresponding R_i and state completion detector (SCD) high. SCD going high will trigger all inputs to low and enable left environment to send a new data. Meanwhile, R_i going high causes right completion detector (RCD) lower, restoring all S_i signals high. and preventing the n-stack gate re-firing even ifa new data arrives. The restoring of all S_i, in turn, resets SCD to low. When right environment trigger all R_i to low, and asking new data to be processed, RCD is going high and n-stack is ready to re-evaluate. Nyström et al. also proposed a dual-rail single track pipeline template based on self-resetting pulsed-logic circuits. But, in general, this template requires more transistors and significantly slower [59, 60].

Part I

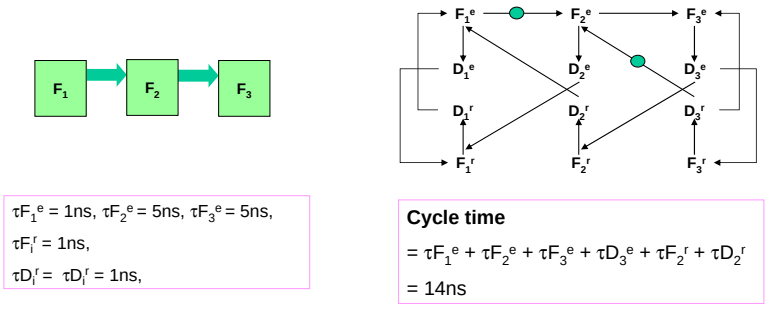
Pipeline Optimization

Chapter 3

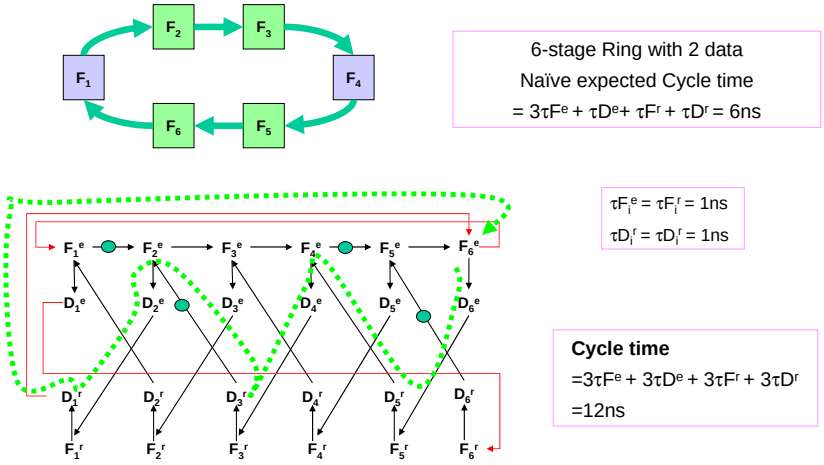
Cycle Time of deterministic Asynchronous Pipeline

A deterministic pipeline is typically partitioned into a set of *stages* that communicate via point-to-point asynchronous channels. Each channel includes a set of wires for data in the forward direction and handshaking control signal going in the reverse direction. Marked graphs are typically used to analyze the performance of an asynchronous circuit in terms of the above quantities [84, 37, 88, 65]. In particular, each cycle in the graph has a *cycle metric* that is the sum of the delays of all associated transitions divided by the number of tokens that can reside in the cycle. The *cycle time* of a deterministic pipeline is defined as the largest cycle metric in its marked graph representation [14, 65]. Note that the cycle time is the inverse of the throughput (measured as average data items processed per unit time).

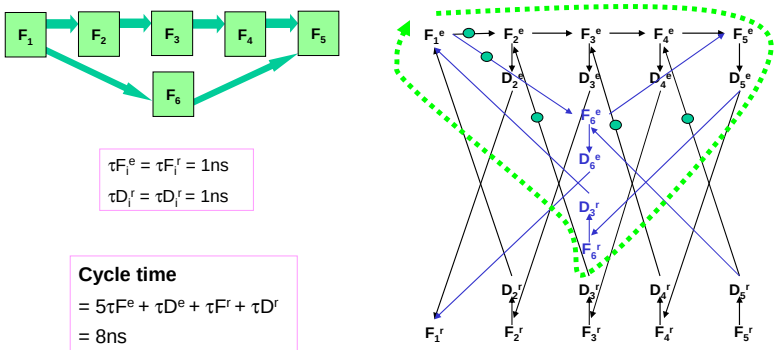
An asynchronous circuit can be structurally modelled with a *circuit model* consisting of one node per stage and a directed edge for every channel. For efficiency



(a) Local pipeline cycle time



(b) Algorithmic cycle time



(c) Structural cycle time

Figure 3.1: Cycle time of asynchronous pipelines.

and clarity, it is useful to realize that the cycles in the marked graph can be partitioned into three classes. The first corresponds to *local pipeline constraints* and corresponds to the interaction of neighboring pipeline stages in the circuit model. For example, consider the simple 3-stage pipeline and corresponding marked graph illustrated in Figure 3.1(a).[1] This marked graph models pipelines using Williams PS0 template style. For this marked graph, there exists three *one-token* cycles, containing only one-token, as follows:

$$\tau(F_1^e) + \tau(F_2^e) + \tau(F_3^e) + \tau(D_3^e) + \tau(F_2^r) + \tau(D_2^r),$$

$$\tau(F_1^r) + \tau(F_2^r) + \tau(F_3^r) + \tau(D_3^r) + \tau(F_2^e) + \tau(D_2^e),$$

$$\tau(F_1^e) + \tau(F_2^e) + \tau(D_2^e) + \tau(F_1^r) + \tau(F_2^r) + \tau(D_2^r$$

The intuition behind the first of the three cycles is as follows. After stage 1 evaluates, stage 2 can evaluate, followed by stage 3. Once stage 3 evaluates, the results from stage 2 are no longer needed and it can precharge. Once stage 2 pre-charges, stage 1 can re-evaluate, completing the cycle. The intuition of the other cycles are similar. The *local cycle time* is the maximum of these delays and, because, there are no loops in the circuit graph, this is also the overall cycle time of the circuit. For the delay parameters given in Figure 3.1(a), the cycle time evaluates to 14ns. Note that for general PS0 pipelines that contain forks and joins the above equations must be modified to include delays associated with additional control overhead.

[1]Note that the places in the marked graphs are omitted for brevity.

The second and third classes map to loops in the undirected version of the circuit model. The second corresponding to *algorithmic loop dependencies* involves sequence of signal transitions that map to directed loops in the circuit and maps to an *algorithmic cycle time*. For example, in asynchronous pipeline rings which implement iterative algorithms, e.g., Williams' asynchronous divider [84], the cycle time may be dictated by how long it takes for a data or bubble (i.e., a single token) to travel around the ring. This concept is abstractly illustrated in the ring of 6 pipeline stages depicted in Figure 3.1(b).

The third class, *fork-join dependencies*, corresponds to sequences of signal transitions that map to other undirected loops in the circuit model corresponding to alternate paths between pairs of circuit stages, i.e, fork-and-join paths. In synchronous pipelines, the number of pipeline stages along different paths between two stages must be the same to ensure correctness. In contrast, this is not guaranteed for correct operation of an asynchronous pipeline. However, a widely unbalanced number of stages in fork-and-join paths may impact performance because one path may prevent data or bubble injection into the other path. These cycles define a *fork-join cycle time* that may also limit circuit performance.

The optimization techniques developed in this work focus on the general class of pipelines consisting of local, algorithmic, and fork-join cycles, where only sequences of up to 3 stages contribute local pipeline constraints. This covers general deterministic circuits using most pipeline strategies of current interest. In addition, we

assert that extensions to pipeline strategies in which fewer than 3 [62, 72, 61] or more than 3 stages yield local pipeline cycles are straight-forward.

Chapter 4

Pipeline Optimization Problem and its Formulation

The abstract circuit models used for analyzing pipelines assume a fixed pipeline structure and thus cannot be directly used as a model to optimize the pipeline structure itself. More specifically, a pipeline optimization model must characterize the set of possible pipeline structures. This chapter describes our proposed model. We first describe a model for the most general pipeline optimization problem and then follow it with a model for an important sub-problem called slack optimization.

4.1 An Abstract Asynchronous Pipeline

If we ignore timing details regarding completion sensing and control circuity, we have observed that the Petri net models for Williams style PS0 scheme [84] (along with another style defined in his thesis called PC0) as well PCHB and WCHB [46]

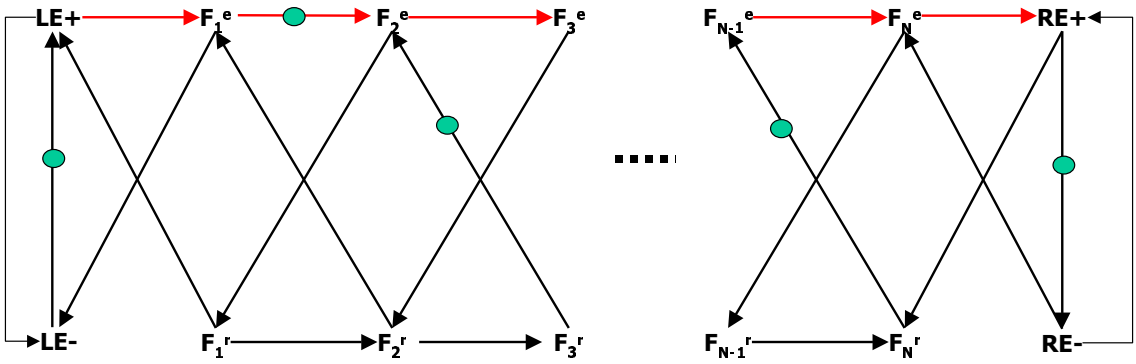


Figure 4.1: Marked graph model of an abstract asynchronous pipeline.

all reduce to that depicted in Figure 4.1. Thus, this new marked graph abstractly models many four-phase pipelines.

In this abstract model, every three pipeline stages has three *one-token* cycles with cycle metrics as follows

$$\tau(F_i^e) + \tau(F_{i+1}^e) + \tau(F_{i+2}^e) + \tau(F_{i+1}^r),$$

$$\tau(F_i^r) + \tau(F_{i+1}^r) + \tau(F_{i+2}^r) + \tau(F_{i+1}^e),$$

$$\tau(F_i^e) + \tau(F_{i+1}^e) + \tau(F_i^r) + \tau(F_{i+1}^r)$$

The basic intuition behind the first of the three cycles is as follows. After stage i evaluates, stage $i+1$ can evaluate, followed by stage $i+2$. Once stage $i+2$ evaluates, the results from stage $i+1$ are no longer needed and it can precharge. Once stage $i+1$ pre-charges, stage i can re-evaluate, completing the cycle. Note that this is a general characteristic of PS0, PCHB, and WCHB based pipelines.

We find this more abstract model useful to analyze the complexity of pipeline optimization as well as describing the functionality of many basic asynchronous pipelines. In fact, based on this observation, we often will constrain our analysis to pipeline schemes whose performance depends on sequences of 3 stages (but perhaps with different cycle metrics than those described above) and argue that this constrained problem covers many practical asynchronous pipeline schemes. Moreover, we assert, that extensions to pipeline strategies in which fewer than 3 or more than 3 stages yield one-token cycles are straight-forward.

4.2 General Pipeline Optimization Model

Our pipeline optimization model is a labeled directed graph (S, U, M, F, L, κ), with nodes S, edges $U \subseteq S^2$, binary labels on edges $M : U \rightarrow \mathcal{B}$, and two sets of binary label on nodes $F : S \rightarrow \kappa$, $L : S \rightarrow \kappa$, and a constant κ. The edges U represent unpartitionable combinational blocks called *units*. The unit u_i has a function evaluation delay $\tau(f_i^e)$, a function reset delay $\tau(f_i^r)$, a completion sensing delay for function evaluation $\tau(d_i^e)$, a completion sensing delay for function reset $\tau(d_i^r)$, a control overhead delay for function evaluation $\tau(c_i^e)$, and a control overhead delay for function reset $\tau(c_i^r)$.

The nodes S represent candidate boundaries between pipeline stages called *slots*. The labels F denote the number of latches pre-assigned by the designer to each slots. For $F(s) \geq 1$, the first latch may be an *abstract latch* that does not represent any

explicit storage element. In particular, many of the Williams PS0 and Caltech's style pipelines [84, 46] need not have explicit latches to separate pipeline stages. Rather, each stage has internal storage for this purpose. Consequently, the first latch in these styles simply delineates a pipeline stage boundary. In contrast, any second or additional latches, must be associated with explicit storage elements. As an example, PS1 is an optimized form of PS0 where each stage is followed by an explicit storage element. We call these explicit storage elements *pipeline buffers*. For convenience, we let κ denote the maximum number of latches assigned to any slot. To simplify the optimization problem it is also possible to limit κ to one and introduce $\kappa - 1$ fictitious units in between every pair of real units.

The labels M denote the edges u_i for which independent data can initially reside. We require that every loop in the pipeline optimization model contain at least one edge that is labeled with a data. However, loops may have multiple, such labeled edges, reflecting the algorithmic intention to have multiple independent data flowing simultaneously through the circuit. Thus, more generally, we require that every loop in the pipeline optimization model be assigned enough abstract latches to support the number of edges u_i labeled with independent data. For example, for both Williams' PS0 and PC0 schemes, the minimum number of latches to support d independent data is $2d + 1$ [84, 74]. Also, we must consider *terminal* slots that have either no incoming or no outgoing edges. To ensure that the cycle time can be computed, we require that terminal slots be pre-assigned abstract latches.

Otherwise it is unclear how to account for the delay of units attached to terminal slots when computing the cycle time. These two conditions together ensure that the cycle time is *well-defined*.

The function evaluation delay of stage$_i$ is defined as $\tau(F_i) = \sum_{u_j \in stage_i} \tau(f_j)$. The reset delay of stage$_i$ is usually defined as $\tau(R_i) = max_{u_j \in stage_i} \tau(r_j)$ based on the assumption that all units within a stage reset (e.g., precharge) simultaneously. The completion sensing delays of stage$_i$ is set to the last unit's completion sensing delay for both function evaluation and reset. The intuition here is that the completion sensing units for the other units are not needed and can be discarded. Similarly, the control overhead delays of stage$_i$ (for both function evaluation and reset) are defined as the number of input control signals to F_i and R_i.

The labels L denote number of abstract latches *to be* assigned to each slot, also referred to as the *latch assignment*. The *min-latch pipeline optimization problem* is to find a minimum cardinality latch assignment that yields a cycle time that is well-defined and less than or equal to a given constraint δ. As an example, Figure 4.2 illustrates the possible outcome of our optimization problem.

Example To make this model more concrete, consider the pipeline optimization model for a Huffman decoder [8] depicted in Figure 4.3 using the PS0 pipeline scheme. The model decomposes the Huffman circuit into 11 units separated by 9 slots and includes the estimated delays for each unit. There are three loops in this

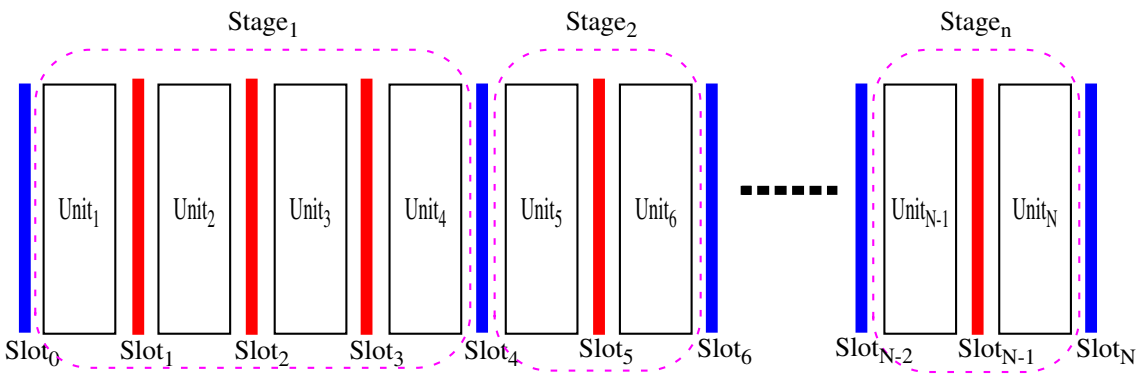


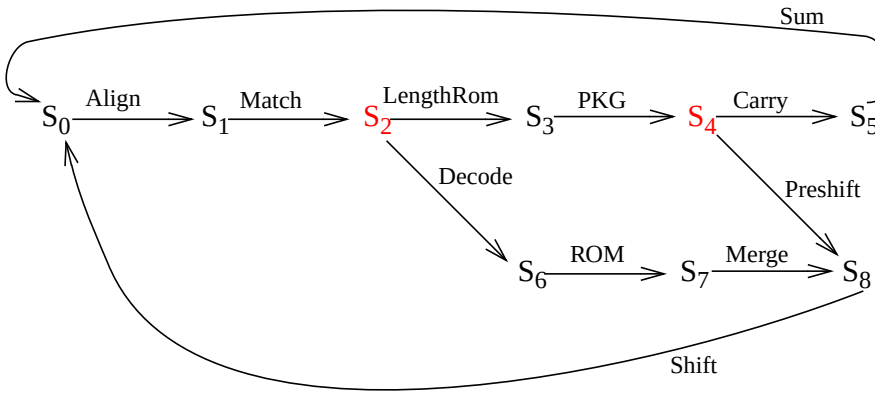
Figure 4.2: Our optimization model of an asynchronous linear pipeline.

optimization model, each representing an algorithmic loop dependency. The maximum sum of the unit evaluation delays along any such loop represents a lower bound on the cycle time. In this case, the evaluation delays of the top loop dominates, yielding a lower bound of 46.[1] □

4.3 Slack Optimization Model

We also identify an important sub-problem of the general pipeline optimization problem in which we consider adding latches to those slots already pre-assigned with at least one latch. In other words, we assume that the degree of functional pipelining is already been fixed and consider only the problem of adding pipeline buffers to improve performance. In particular, we define *slack optimization* as finding a minimum number of additional pipeline buffers required to yield a cycle time that is well-defined and less than or equal to a given constraint δ.

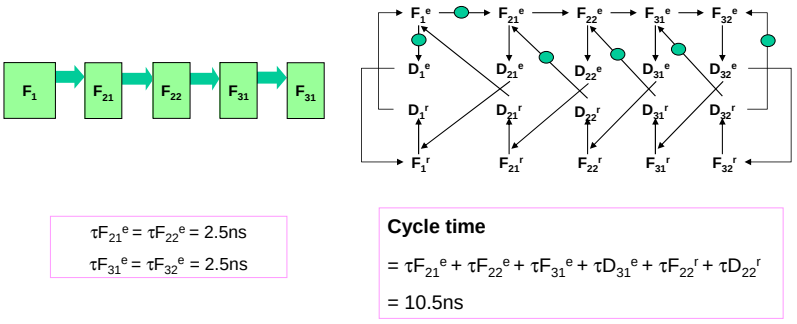
[1]Thus, our optimization problem is to find a minimum abstract latch assignment that yields a cycle time of no larger than 46.



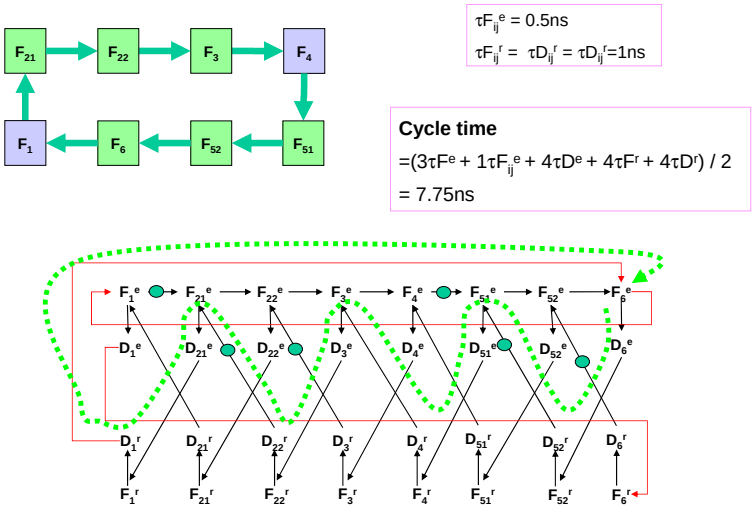
	Align	Match	Length Rom	PKG	Carry	Sum	Decode	ROM	Merge	PreShift	Shift
$\tau(f^e)$	3	10	5	3	3	3	10	15	5	3	3
$\tau(f^r)$	2	3	0	2	2	2	3	0	2	1	1
$\tau(d^e)$	2	6	2	2	2	2	2	2	2	2	2
$\tau(d^r)$	2	6	2	2	2	2	2	2	2	2	2
$\tau(c^f)$	0	0	2	0	2	0	2	0	0	2	0
$\tau(c^r)$	0	0	2	0	2	0	2	0	0	2	0

Figure 4.3: An asynchronous Huffman decoder model and its detailed delay information.

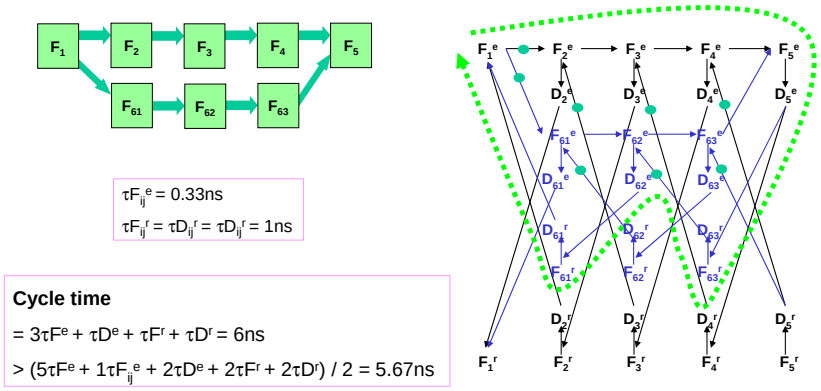
Figures 4.4 and 4.5 illustrate how the various forms of performance bottlenecks described in Figure 3.1 can all be resolved by either splitting existing pipeline stages or adding new pipeline buffers, respectively.



(a) Improving the throughput of the linear pipeline depicted in Figure 3.1(a) by pipelining slow stages.

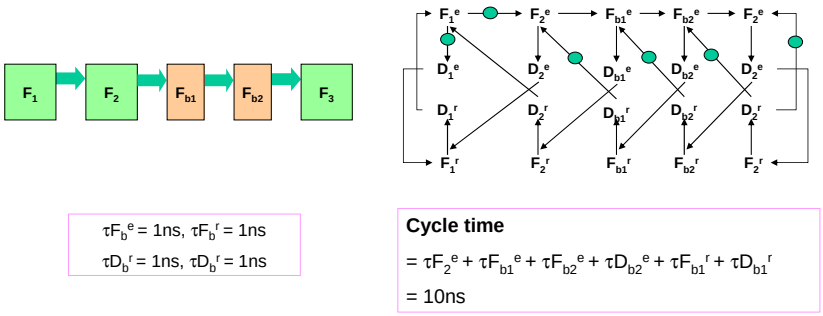


(b) Improving the throughput of the small ring depicted in Figure 3.1(b) by pipelining two stages, increasing the number of bubbles in the ring.

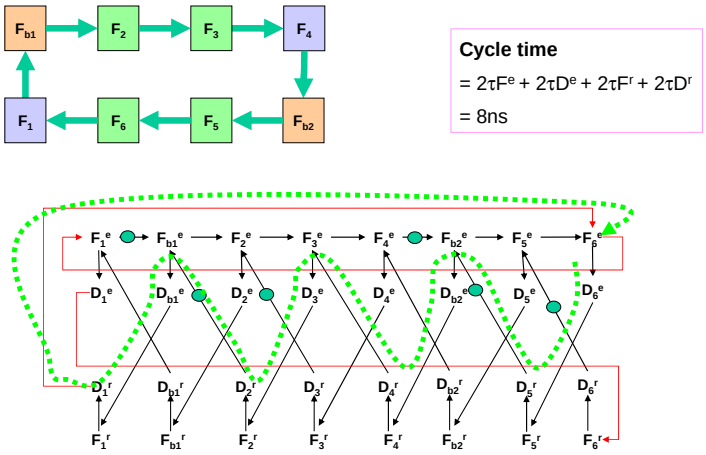


(c) Improving the throughput of the fork-and-join structure in Figure 3.1(c) by pipelining the short branch.

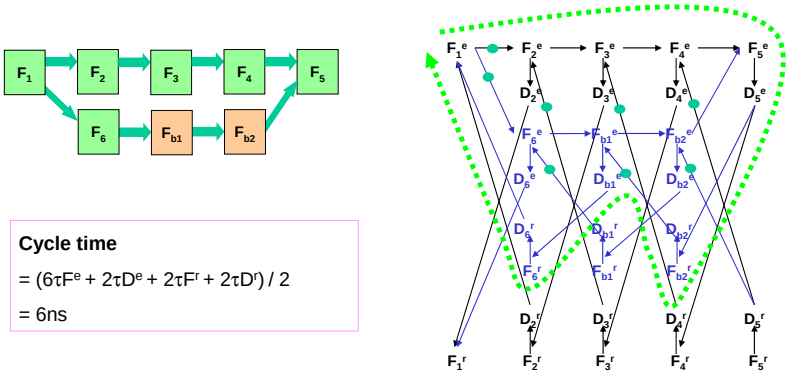
Figure 4.4: Asynchronous pipeline optimization by inserting abstract latches.



(a) Improving the throughput of the linear pipeline depicted in Figure 1(a) by adding pipeline buffers.



(b) Improving the throughput of the small ring depicted in Figure 1(b) by adding two pipeline buffers, increasing the number of bubbles in the ring.



(c) Improving the throughput of the fork-and-join structure in Figure 1(c) by adding two pipeline buffers to the short branch.

Figure 4.5: Asynchronous pipeline optimization by pipeline bufers.

Chapter 5

Complexity Analysis

Given that the basic synchronous retiming problem can be optimally solved in polynomial time [44], it seems prudent to determine the complexity of our problem before exploring efficient algorithms. This chapter proves that our problem is NP-complete for the simplified pipelining performance model depicted in Figure 4.1. This graph is equivalent to the more complicated marked graph in Figure 2.5(b), 2.6(b), and 2.7(b) for the special case of $\tau(f_i^r) = \tau(f^r) = \tau(d_i^e) = \tau(d_i^r) = \tau(c_i^e) = \tau(c_i^r) = 0$, for all i units. Lastly, we assume that the given cycle time constraint δ is larger than cycle metrics associated with loop dependencies, which for this simplified dependency graph model, is independent of the degree of pipelining. The proof of NP-completeness for a variety of more complex marked graphs, including the graph depicted in Figure 2.5(b), 2.6(b), and 2.7(b), then follows directly by *restriction* [30]. The intuition behind these results is that, in general, the number of potentially-optimal pipeline configurations in an asynchronous circuit is much larger than considered by synchronous retiming for a similar-sized problem.

5.1 Complexity Analysis of Asynchronous Pipeline Optimization Problem

We define the *Asynchronous Pipeline Decision (APD)* problem as the task of determining whether there exists a pipelining strategy using K or less abstract latches for which the pipeline cycle time is well-defined and less than or equal to δ. We prove this problem is NP-complete by reduction to 3SAT problem in two steps.

First, let Z be a set of variables z_i and X be a collection of sum-of-product clauses over positive and negative literals of Z such that each clause $x_i \in X$ has $|x_i| = 3$ [30]. The 3-Satisfiability (3SAT) problem is a well known problem whose task is determine whether there exists a satisfying truth assignment for X. The complexity of the 3SAT problem has been well established:

Theorem 2 Complexity of 3-Satisfiability (3SAT) *[30]*

3SAT problem is NP-complete.

Consider a simplified pipeline optimization model $G = (S, U)$, where S is a set of slots, U is a set of units, and no cycle consists of less than three slots. We define a *3U1L assignment* as the task of determining whether there exist a set of slots $S' \subset S$ with cardinality less than or equal to K, for which every terminal slot is in S' and every three consecutive unit sequence should span at least one slot in S'. The first step of our proof involves showing that the 3U1L problem is NP-complete.

To do this, we follow the same reduction strategy to 3SAT from the vertex cover problem [30]. We observed that ensuring every three unit sequence is spanned by at least one slot in S' is equivalent to ensuring that every middle unit is touched by at least one slot in S'. Mapping units to edges and slots to vertices, this is equivalent to ensuring that all middle edges must be covered by selected vertices, which is the key point behind the following proof.

Lemma 1 Complexity of 3U1L Assignment (3U1L)

The 3U1L problem is NP-complete.

Proof (Sketch) First, the 3U1L problem is in NP because a modified depth-first-search algorithm can verify that every that every terminal slot is in S', every three unit sequence contains a slot in S', and that S' is the appropriate size in polynomial time. To prove 3U1L is NP-hard, we show that our problem can be reduced to the 3SAT problem which is known to be NP-complete.

We first construct a graph $G = (S, U)$ and a positive integer $K \leq |S|$ such that G has a 3U1L assignment with K or less latch assignment if and only if X is satisfiable. The graph consists of three different subgraphs. First, for each variable $z_i \in Z$, we create a *truth-setting* subgraph $T_i = (S_i, U_i)$ with $S_i = \{t_i, z_i, \bar{z}_i, \bar{t}_i\}$

and $U_i = \{\{t_i, z_i\}, \{z_i, \bar{z}_i\}, \{\bar{z}_i, \bar{t}_i\}\}$. For each clause $x_j \in X$, there is a *satisfaction-testing* subgraph $A_j = (S'_i, U'_i)$, consisting of three slots and three units joining them to form a cycle with three slots.

$$S'_j = \{a_1[j], a_2[j], a_3[j]\}$$

$$U'_j = \{\{a_1[j], a_2[j]\}, \{a_2[j], a_3[j]\}, \{a_3[j], a_1[j]\}\}$$

The third and last subgraph consists of only *communication* units and is the only subgraph that depends on which literals occur in the clauses of the 3SAT problem. For each clause $x_j \in X$, let the three literals in x_j be denoted by p_i, q_i and r_i. Then, let the communication units of A_j be given by

$$U''_j = \{\{p_j, a_1[j]\}, \{q_j, a_2[j]\}, \{r_j, a_3[j]\}\}$$

The construction of our instance of 3U1L is composed by setting $K = 3|Z| + 2|X|$ and $G = (S, U)$ where S is an union of all S_i and S'_j and U is an union of U_i, U'_j and U''_j. Note, that this construction clearly has polynomial time complexity.

Now, we show that the original 3SAT problem is satisfiable if and only if the constructed 3U1L problem is satisfiable. First, suppose that $S' \subseteq S$ is a valid solution of 3U1L for G with $|S'| \leq K$. S' must contain *at least* three slot from each T_i and *at least* two slots from each A_j. Since $K = 3|Z| + 2|X|$, however, we can further conclude that S' must contain *exactly three* slots from each T_i, two of

which are terminal slots, and *exactly two* slots from each A_j. Note that the third (non-terminal) slot chosen in each T_i defines which variable, z_i or $\bar{z}_i$, is set to one in the solution to the 3SAT problem. To see how this truth assignment satisfies each of the clauses $x_j \in X$, consider the three units in U''_j. Exactly one of these three units must not be attached to a slot in $S' \cap A_j$ because only two of the three slots in A_j can be in S'. This slot thus must be connected to a slot z_i ($\bar{z}_i$) that is in S' which implies that the clause x_j is satisfied. For the other direction, suppose a truth assignment satisfies X. The corresponding 3U1L solution S' contains three slots from each T_i, two of which are the terminal slots and one defined by the truth assignment, and two slots from each A_j, corresponding to the slots not connected to the third (non-terminal) slot of T_i. This set of selected slots ensures that every three consecutive unit sequence has at least one selected slot.

Figure 5.1 shows an example of the proposed constructed graph for the 3SAT problem $Z = \{z_1, z_2, z_3, z_4\}$ and $X = \{\{z_1, \bar{z}_3, \bar{z}_4\}, \{\bar{z}_1, z_2, \bar{z}_4\}\}$. For example, the 3U1L solution $S' = \{z_1, z_2, \bar{z}_3, \bar{z}_4,\ a_2[1], a_3[1], a_1[2], a_3[2]\}$ identifies the satisfying solution to the 3SAT problem $z_1 = 1, z_2 = 1, z_3 = 0$, and $z_4 = 0$.

The second step of the proof requires the following useful definitions. We define a sequence of units to be *decomposed* into k stages by a slot assignment if the units are part of k distinct stages (as defined by the slot assignment). We say a sequence of units is a *violating unit sequence (VUS)* if the sequence must be decomposed into at least 4 stages in order to satisfy the cycle time constraint, δ, i.e., there doesn't

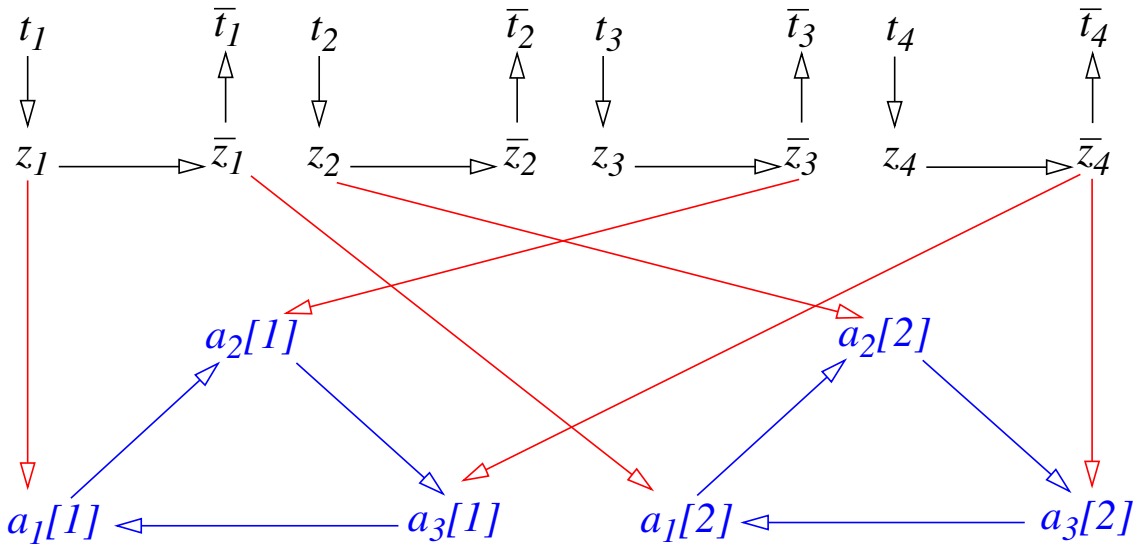


Figure 5.1: A 3U1L instance resulting from a 3SAT instance.

exist any slot assignment that yields a well-defined cycle time less than or equal to δ that decomposes the sequence of units into 3 or fewer stages. We say a sequence of slots is a *violating slot sequence (VSS)* if it is spanned by a VUS, i.e., their exists a VUS that connects the sequence of slots.

Theorem 3 *The pipeline cycle time is less than or equal to δ if and only if every VUS spans parts of at least 4 stages, i.e., contains units in 4 distinct stages. In other words, the corresponding VSS must contain at least 3 abstract latches.*

Proof (Sketch) $\Leftarrow$: We first prove that if every VUS spans at least 4 stages, the cycle time constraint δ is satisfied. To prove this, we prove the equivalent statement that if the cycle time constraint δ is not satisfied, there must exist a VUS which constitutes at most 3 stages. To see this, note that to violate δ, there must exist at least three consecutive stages whose cycle time is larger than δ. The sequence of

units that correspond to this sequence of stages is a VUS, thereby completing this part of the proof.

$\Rightarrow$: If cycle time constraint δ is satisfied, every VUS constitutes parts of at least 4 stages. We prove the above statement by contradiction. Assume that that cycle time constraints δ is satisfied but that there exists a VUS with three or less stages. By the definition of pipeline cycle time, this VUS however implies that δ is violated, a contradiction.

Finally, we prove NP-completeness of APD problem by restricting the APD problem such that $\tau(f_i^e) = 0.2$ and $\delta = 0.99$ and showing a reduction to the 3U1L problem.

Theorem 4 *The APD problem is NP-complete.*

Proof (Sketch) We first show that APD problem $\in NP$. To verify that a given solution π to the APD problem is valid, we must verify that it has less than or equal to K slots and that it yields a circuit whose cycle time satisfies the given cycle time constraint δ. The first part involves counting the number of slots in π and the second part of the problem involves finding the longest sequence of three stage delays which can be solved using a trivially modified version of depth first search. Thus, both of these steps take polynomial time.

Next, to prove the APD problem is NP-hard, we provide a polynomial-time algorithm that maps any instance of the 3U1L problem to an instance of the APD problem. First, we construct an APD problem instance G' from an instance of

3U1L problem. Every unit u_i in G is divided into two unit $u_{i,1}$ and $u_{i,2}$ in G'. Moreover, for each new slot created, we create two additional slots and add units in between the three slots to make a directed ring of size 3. Thus, G' consists of $5|U|$ units and $|S| + 3|U|$ slots. The transformation from G to G' can be done easily in polynomial time.

Next, we prove that there exists a subset of slots with cardinality less than or equal to K latches that satisfies any instance of the 3U1L problem if and only if there exists a latch assignment using $K' = K + 3|U|$ that satisfies the constructed instance of the APD problem.

Lets consider both directions of the if and only if condition. First, suppose there exists a latch assignment with K latches that satisfies the 3U1L problem. We observe that a property of our construction is that every five unit sequence in G' has a corresponding 3 unit sequence in G. In particular, every five unit sequence in the constructed graph G' consists of two newly added slots and two slots that were consecutive in G, one of which must be in the solution to the 3U1L problem. Consider the slot assignment in which, in addition to the selected latches in the 3U1L solution, every newly added slot is assigned a latch. First, notice that this assignment requires less than or equal to $K + 3|U|$ latches. Second, notice that solution guarantees that every five unit sequence in the constructed graph spans three latches, that the cycle time is well-defined and is less than or equal to δ.

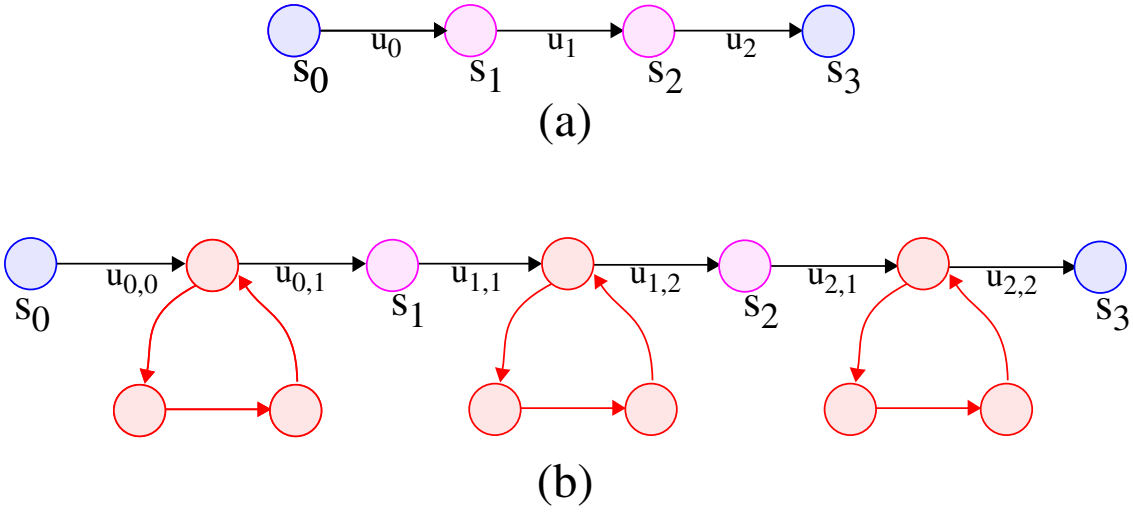


Figure 5.2: An example of mapping a 3U1L problem instance to an APD problem instance.

Conversely, suppose there exists a satisfying latch assignment using less than or equal to $K' = K + 3|U|$ latches for an instance of APD problem. Another property of our construction is that every three unit sequence in G has two corresponding five unit sequences in G'. Each corresponding five unit sequences spans two slots that were consecutive in G and two newly created slots. Any solution to the APD problem must assign a latch to one of the consecutive slots in G. Consider the solution to the 3U1L problem created by selecting these slots in G. Each three unit sequence in G spans a selected slot and the number of selected slots must be less than or equal to K, thereby completing the proof.

An example of mapping a 3U1L problem to an APD problem is depicted in Figure 5.2.

5.2 Complexity Analysis of Slack Optimization Problem

First we assume that $\tau(f_i^e) = \tau(f_i^r) = \tau(d_i^e) = \tau(d_i^r) = \tau(c_i^e) = \tau(c_i^r) = 0$, for all pipeline buffer. We define the *Slack Decision (SD)* problem as the task of determining whether there exists a slack optimization strategy using K or less pipeline buffers for which the pipeline cycle time is less than or equal to δ. We prove the complexity of the SD problem is NP-complete by restriction.

Finally, we prove NP-completeness of SD problem by restriction of the SD problem such that $\tau(F_i^e) = 0.2$ (which means function evaluation delay of every stage before slack optimization is 0.2) and $\delta = 0.59$.

Theorem 5 *The SD problem is NP-complete.*

Proof (Sketch) By restriction. Allow maximum one pipeline buffer for each stage. Now the SD problem itself is exactly 3U1L problem.

Chapter 6

Optimal Algorithm

There exists a variety of techniques that may be used to solve our minimization problem. The most general technique is to cast the problem as an integer programming problem and use generic IP solvers. Alternatively, one could define a binary decision diagram (BDD) [33] describing the possible solutions for each VSS and take the product of all such BDDs. Any path through the BDD that leads to one represents the candidate of a valid solution, and the path with the minimal number of "1" branches represents a candidate for the minimal solution [73]. Both of these solution strategies, however, do not take advantage of the structure of the solution space and thus may be inefficient. In contrast, this chapter proposes an efficient branch and bound algorithm that incorporates a new lower bound technique tailored to our problem. Moreover, we assert that our branch and bound algorithm is more robust than possible BDD-based techniques because it may be terminated early to obtain a non-optimal solution, whereas BDD-based approaches may catastrophically fail if the BDD-size blows up.

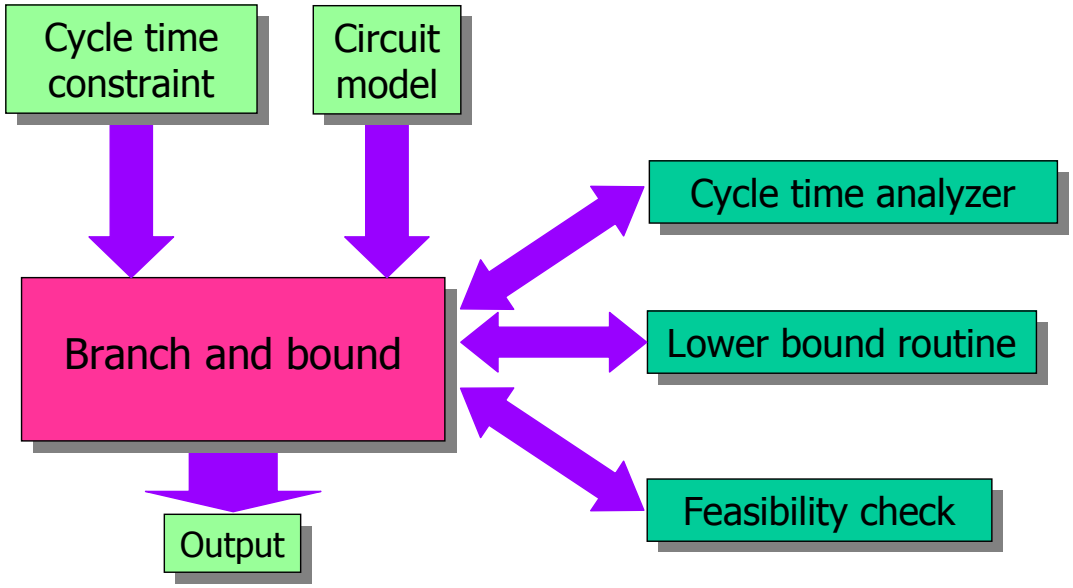


Figure 6.1: Overview of APO

Our asynchronous pipeline optimizer (APO) consists of a branch and bound framework that interacts with a cycle time analyzer to check for algorithmic constraints as well as both a lower bound routine and a feasibility checker to prune branches off the decision tree. The overview of APO is depicted in Figure 6.1 The theoretical foundation of a satisfying node is based on the notion of properly decomposed VUSs as explained in the subsequent section. Subsequent sections describe the constitute algorithms in detail.

6.1 Proper Decomposition of Violating Unit Sequence

To solve the general optimization problem, we first introduce the following definitions. A VUS is *Properly Decomposed (PD)* by a slot assignment if the following conditions are satisfied:

Condition 2 Covering condition*: The VUS is decomposed into at least* 4 *stages by the slot assignment.*

Condition 3 Satisfying condition*: The VUS does not contain any (complete) sequence of stages which violate* δ*.*

Let M be a set of VUS such that every sequence of units is either a subset of a VUS in M or a superset of a VUS in M, that is no sequence can just partially intersect or disjoint with all VUS in M.

Lemma 2 *The local pipeline cycle time is met if and only if all VUS* $\in M$ *are properly decomposed.*

Proof (Sketch) $\Leftarrow$: Consider a 3-stage sequence of units, which violates the cycle time. It is either a superset or a subset of at least a VUS in M. If it is the superset of a VUS in M, it can't be a 3-stage or less sequence. (Contradiction of the condition 2). If it is the subset of a VUS in M, it should be properly decomposed. (Contradiction of the condition 3).

$\Rightarrow$: Proof by the definition of VUS.

The key theorem that identifies our optimization approach follows directly from the above lemma.

Theorem 6 *The local pipeline cycle time is met with the minimum abstract latches if and only if all $VUS \in M$ are properly decomposed with the minimum slot assignment.*

6.2 Branch and Bound Algorithm

The nodes in our branch and bound tree represent slots. Each node has up to two children, one representing the partial solution in which the slot is assigned an abstract latch, referred to as a *slot-assigned-child*, and the other representing the partial solution in which the slot is not assigned an abstract latch, referred to as a *slot-excluded-child*. Each node is associated with the set of VSSs that contain that slot. Each time a new abstract latch is added to a partial solution, we compute the subset of associated VSSs that are properly decomposed and modify marked graph representation. When all VSSs are properly decomposed, we analyze marked graph to verify that cycle metrics associated with other dependencies are less than cycle time constraints δ. We do not search the subtree routed at a slot-assigned-child when 1) the number of abstract latches assigned up to that child node plus the derived lower bound for that subtree is larger or equal to the current best solution or 2) the child node represents a solution better than the current best, in which

case the current best solution is updated, or 3) the cycle metrics associated with any loop dependence involving only functional evaluation delays exceeds δ.[1] We do not search the subtree routed at a slot-excluded-child when we determine there exist no feasible solution for a VSS associated with the slot.

Nodes associated with slots assigned with the least number of abstract latches are searched first and nodes associated with slots that have been excluded in a parent are never searched. We break ties between nodes that represent slots that have equal number of assigned abstract latches are broken by prioritizing the slot that is associated with the most uncovered VSSs computed once at the beginning of the search. We prioritizes assigning abstract latches over pipeline buffers, thereby greedily avoiding the increased latency of pipeline buffers.

6.3 Lower Bound Heuristic

In the traditional branch and bound approaches to covering problems, the MIS_QUICK independent-set-based lower bound algorithm [33] is widely used because its simple and fast. We generalize this algorithm to our optimization problem as follows. For each node in the branch and bound tree, we create a *lower bound graph* consisting of a vertex for each VSS and an edge between every two VSSs that share at least one slot. Each vertex is labeled with the number of additional abstract latches

[1]This last condition is because additional abstract latches cannot decrease cycle metrics associated with data limited loop dependencies.

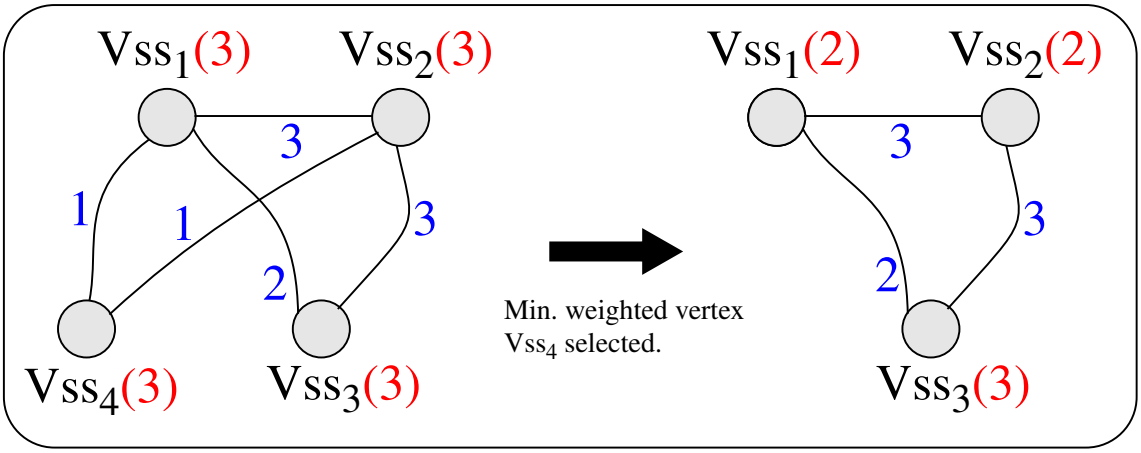


Figure 6.2: An example of the lower bound heuristic.

needed to be assigned for the VSS to be satisfied (which, we recall, is only one of two conditions to be properly decomposed). Each edge is labeled with the number of slots shared between the two VSSs. We define the *weight* of a vertex as the sum of connected edge labels divided by the vertex label. We identify the vertex with the minimum weight and decrease all connected vertices by the minimum of the identified vertex's label and the connecting edge label. We then remove the identified vertex along with all connected edges and iterate. It can be easily verified that the sum of the identified verticies' labels is a lower bound of our problem. Figure 6.2 shows an example of one iteration of our lower bound heuristic.

6.4 Cycle time analysis

In this section, we describe an efficient method based on Karp's algorithm to find the largest cycle metric of potential solutions. The time complexity of the traditional

approach proposed by Ramamoorthy [65] is $O(P^3)$. The naive application of Karp's algorithm [40, **?**] to our problem is to create a vertex for each marked place and to create edges between two vertices if there exists a path between their corresponding places. The weight of an edge is the largest sum delay of such paths. Karp's algorithm will then find the maximum mean cycle which is equivalent to our largest cycle metric. To improve the time complexity of this procedure, we propose a new conversion procedure described in Figure 6.3.

The key idea of our procedure is to iteratively remove any vertex associated with an unmarked place by bypassing the vertex with edges from incoming vertices to outgoing vertices with a corresponding weight. Interestingly, the resulting reduced di-graph may have more *simple* cycles than the original which at first glance may suggest that the maximum mean cycle is not preserved. However, all additional simple cycles originate from non-simple cycles and consequently are guaranteed to have metrics no larger than the maximum mean cycle. Thus, as proven in Lemma 3, the graph reduction preserves the largest cycle metric.

Lemma 3 *Any cycle C'' in the di-graph which does not correspond to a simple cycle in the marked graph is not a maximum mean cycle of the di-graph.*

Proof (Sketch) Let the sum of the delay along the i^{th} simple cycle C_i be $D(C_i)$ and the number of tokens along C_i be $T(C_i)$. Let the cycle corresponding to the largest cycle metric in the marked graph be denoted C_k.

Construct di-graph{
  /* Construct di-graph for the cycle time analysis */
  for each place p_i
    create a vertex v_i
  for each place p_i
    for each place p_j
      if p_i and p_j are connected through a transition t_k
        Create edge e_{ij} and $w_{ij} = d(t_k)$
}
Reduce di-graph{
  /* Reduce the number of vertices in the di-graph */
  for each vertex v_i which is corresponding to unmarked place p_i
    for each incomming vertex v_a to v_i
      for each outgoinging vertex v_b from v_i
    create a edge e_{ab} and $w_{ab} = w_{ai} + w_{ib}$
  remove vertex v_i along with its adjacent edges
}

Figure 6.3: The Conversion procedure to di-graph for the cycle time analysis

The cycle C'' exists if and only if there exists a corresponding non-simple cycle in the marked graph. This non-simple cycle can be divided into multiple simple cycles $C_m ... C_n$. Then the mean cycle of $C'' = ((D(C_m)+...+D(C_n))/(T(C_m)+...+T(C_n))$. Because, $D(C_k)/T(C_k) > D(C_l)/T(C_l)$ $(l = m..n)$, the mean cycle of C'' is no larger than $D(C_k)/T(C_k)$.

The time complexity of the proposed graph transformation is $O(P^2)$. The subsequent Karp algorithm takes $O(V \cdot E) = O(P'^3)$, where V is set of marked places in detailed marked graph, E is the set of edges between marked places.

6.5 Experimental Results

We have implemented our algorithm in C. To demonstrate its feasibility and limitations, we applied it to the asynchronous Huffman decoder model depicted in Figure 4.3 as well as three scalable asynchronous circuit structures, a linear pipeline, a pipeline ring, and a pipelined ring-of-ring structure. We tested linear pipelines and pipeline rings with 15, 20, 25 and 30 slots. The last structure (LFSR) we tested with complicated interacting rings containing 5 data tokens. For all examples, we choose Williams' PS0 pipeline scheme. For all scalable examples, the function evaluation delay, the function reset delay, the completion sensing delay for evaluation and the completion sensing delay for reset are randomly generated between 10.0 and 30.0, 5.0 and 15.0, 1.0 and 20.0, and 1.0 and 10.0, respectively.

Table 6.1 and 6.2 shows the experimental results of our algorithm with and without the lower bound algorithm (presented in Section 6) enabled. When the lower bound algorithm is enabled, the run time is cut by half. The results demonstrate that using our lower bound algorithm, the optimal pipeline configuration for moderately-sized problem is feasible. It is also important to note that for large systems, the run-time can be reduced by either removing slots from consideration or pre-assigning slots with abstract latches. For instance, we ran additional experiments where for each structure, we pre-assigned several selected slots with abstract latches. As shown in Table 6.1 and 6.2, the run-times are significantly reduced.

Table 6.3 shows that interesting results of asynchronous slacking matching. To achieve the local pipeline cycle time (20), only a few fast pipeline buffers are required. For example, initially, a path has 17 stages and the other path has 3 stages. Inserting fast 8 pipeline buffers is enough to achieve its local cycle time as a cycle time.

Total # Slots	Total # Units	# PA Latches	Opt. # Latches	Cycle Time	Run Time$_1$ (sec) / # visited branches$_1$	Run Time$_2$ (sec) / # visited branches$_2$
Asynchronous Linear Pipeline ($\delta = 150$)						
15	14	2	8	123.64	0.65 / 2281	0.55 / 1797
20	19	2	11	132.19	19.98 / 47691	18.33 / 37980
25	24	2	13	141.46	211.65 / 529141	172.46 / 317467
30	29	2	15	143.31	2532.11 / 5829341	1325.17 / 2138475
30	29	7	16	146.28	178.28 / 293358	148.77 / 216345
Asynchronous Ring ($\delta = 150$)						
15	15	4(2)	8	142.60	0.18 / 571	0.16 / 466
20	20	6(3)	11	146.45	1.52 / 3346	1.27 / 2664
25	25	8(4)	14	144.99	16.87 / 25018	12.93 / 19346
30	30	10(5)	17	145.80	120.33 / 123979	98.7 / 97374
Asynchronous Huffman Decoder ($\delta = 60$)						
9	11	4	5	54	0.03 / 13	0.01 / 13

Table 6.1: Experimental results for asynchronous pipelines, rings and Huffman decoder. Quantities with a subscript 1 refer to experiments with the lower bound disabled, while quantities with a subscript 2 refer to experiments with the lower bound enabled.

Total # Slots	Total # Units	# PA Latches	Opt. # Latches	Cycle Time	Run Time$_1$ (sec) / # visited branches$_1$	Run Time$_2$ (sec) / # visited branches$_2$
Asynchronous Linear Pipeline ($\delta = 120$)						
15	14	2	10	117.10	0.73 / 1696	0.76 / 1589
20	19	2	13	119.99	6.75 / 16748	5.95 / 11155
25	24	2	16	119.98	63.85 / 119240	57.81 / 80350
30	29	2	19	119.98	604.02 / 1000494	551.37 / 534454
30	29	7	20	118.29	105.90 / 149800	99.46 / 87308
Asynchronous Ring ($\delta = 120$)						
15	15	4(2)	-	-	-	-
20	20	6(3)	-	-	-	-
25	25	8(4)	17	118.14	16.23 / 18821	15.15 / 16182
30	30	10(5)	21	117.08	79.95 / 76839	78.95 / 60704
Asynchronous Huffman Decoder ($\delta = 46$)						
9	11	4	6	46	0.05 / 18	0.01 / 18

Table 6.2: Experimental results for asynchronous pipelines, rings and Huffman decoder. Quantities with a subscript 1 refer to experiments with the lower bound disabled, while quantities with a subscript 2 refer to experiments with the lower bound enabled.

Initial # Stages$_A$	Initial # Stages$_B$	Opt. # Stages$_B$	Cycle Time	Run Time	Opt. # Stages$_B$	Cycle Time	Run Time
		$\delta = 27$			$\delta = 25$		
8	2	3	26	0.07	4	22.5	0.45
13	2	4	26.5	0.55	5	22.8	3.84
17	3	6	26.4	37.78	7	24	132.85
		$\delta = 23$			$\delta = 20$		
8	2	4	22.5	0.42	6	20	4.4
13	2	5	22.8	6.06	7	20	20.5
17	3	9	21.71	715.37	11	20	2125.48

Table 6.3: Experimental results for asynchronous fork-and-join pipelines to demonstrate slack optimization.

Chapter 7

Conclusions

In this work, we formalizes a new asynchronous pipeline optimization problem common to a variety of pipelining styles and proves that it is NP-complete. It then proposes an efficient branch and bound algorithm for the exact solution. The experimental results suggest that the algorithm is feasible for moderately sized systems. Moreover, complexity reduction methods for its application to larger systems are also presented and evaluated.

There are many interesting future directions for this research. For example, although the algorithm as described is restricted to models that do not exhibit choice, the approach can also heuristically be applied to systems with choice modeled by, e.g., free-choice Petri nets. The idea is to sequentially apply the algorithm to distinct choice-free behaviors (e.g., marked graph components) from those with highest probability to those with lowest probability. Specifically, the abstract latches assigned in one iteration would be assumed pre-assigned for the remainder of the optimization process. Other more effective strategies may also be possible and are

an interesting area of future research. In addition, extensions that allow stochastic delays may also be possible and useful.

Part II

Petri Net Reduction

Chapter 8

TSE bounds and evaluating the bounds

The key idea of the bounding approach is to partition an infinite timed execution into a sequence of so called *segments*. These segments are independent and identically distributed (iid). The targeted TSEs are then analyzed within individual segments independent of remaining segments. Since the analysis is done using limited history, it yields upper and lower bounds on the TSEs (via longest path analysis) instead of exact values of the TSEs. These bounds are iid, which facilitates the estimation of the statistics of the bounds with well-known statistical methods [88].

8.1 Partitioning infinite timed executions

Let $\pi = (N_\pi, d, \ell)$ be a timed execution of Σ. It is known that every (untimed) reachable marking of Σ induces a cut ξ (a set of instanced places) which partitions the event graph N_π [11]. The portion of the event graph in between two different

cuts due to the same reachable marking M is a *segment*. Formally, if σ is a firing sequence that starts from a cut ξ and ends at another cut ξ' such that $\ell(\xi) = \ell(\xi') = M$, then the portion of N_π between ξ and ξ' is a segment, denoted by $S(\xi, \sigma)$. A segment $S(\xi, \sigma)$ can also be denoted by $S(\xi, \theta)$ such that $\theta = \{(s, t) | s \in T_s, t \in T_\pi\}$ is a strict partial order of event pairs where T_s is a set of all events in the segment. The segment is *minimal* if it does not contain any other segment starting from ξ.

One simple property of a random time execution of a Petri net considered in this section is that the structures of its segments are independent of each other. This is because the structure of a segment is determined by the choices made on the places inside the segment and choices made in different segments are independent. As a result, the sequence of segments generated by a random timed execution has the property that their structures are not determined by the location of the segment in the sequence. In fact, they are independent and identically distributed (iid). This fact allows us to reason about an infinite execution by considering all possible finite executions of as little as one segment in length.

8.2 Bounding a single TSE instance

We obtain bounds on a TSE instance belonging to a particular segment by ignoring the history of the segment. In other words, our bounds on a TSE instance are defined by assuming tokens in the source places of the segment can be available at any time within $(-\infty, \infty)$. This subsection describes a method to compute the

upper bound using longest path analysis. The lower bound can be computed using duality.

To compute the upper bound, we identify a set of *reference events* that serve the synchronization points for the targeted events. By assuming each of the synchronization points to be critical, one obtains a set of time separations of the event pair. The upper bound is simply the largest separation obtained. To detail this idea, consider upper bounding the TSEs due to separation triple $(s, t, \varepsilon = 0)$. Extension to the case where $\varepsilon \neq 0$ is not difficult.

Let ρ be a path in the event graph of a timed execution $\pi = (N_\pi, d, \ell)$. The set of all paths leading from x to y is denoted by $\mathcal{P}(x, y)$ where $x, y \in P_\pi \cup T_\pi$. A *reference set* for event e of π is a subset of events of π such that every path from a source place of N_π to e contains at least one event in R, and every event in R has a path to e.

It follows from the timing relation (2.1) that if x has a path to y, then y must occur after x by at least the the sum of place delays along any path from x to y. That is, whenever $\mathcal{P}(x, y) \neq \emptyset$,

$$\tau(x) + \max_{\rho \in \mathcal{P}(x,y)} \delta(\rho) \leq \tau(y). \tag{8.1}$$

where $\delta(\rho) = \sum_{p \in \rho} d(p)$. In particular, we say event x is *critical* for event y if (8.1) holds in equality. Further, if R be a reference set for event y. Then, the occurrence

time of y is uniquely determined by the occurrence times of the events of R plus the delay values of places following these events. That is,

$$\tau(y) = \max_{x \in R}[\tau(x) + \max_{\rho \in \mathcal{P}(x,y)} \delta(\rho)]. \tag{8.2}$$

The term $\max_{\rho \in \mathcal{P}(x,y)} \delta(\rho)$ in (8.2) measures the maximum delay on any path from event x to y. For convenience, we write:

$$\delta^*(x, y) = \max_{\rho \in \mathcal{P}(x,y)} \delta(\rho), \tag{8.3}$$

where $\delta^*(x, y) \stackrel{\triangle}{=} -\infty$ if there is no path from x to y, i.e., $\mathcal{P}(x, y) = \emptyset$. For completeness, we define an event e itself to be a path of delay 0, and consequently, $\delta^*(e, e) = 0$.

Suppose the m-th TSE instance starts in the l-th segment of π denoted by $S^{(l)}$. Since the set of source places of a segment is a cut of N_π, its poset must contain a reference set for every event e of segment $S^{(l)}$ (in fact, for every event in segments $S^{(l')}$ if $l' \geq l$). For convenience, if event $e \in S^{(l')}(l' \geq l)$, let us denote by $R(e, l)$ such a reference set. An upper bound ${U_\gamma}^{(m)}(s, t, 0)$ on TSE $\gamma^{(m)}(s, t, 0)$ is determined by (8.4) ([88]).

$${U_\gamma}^{(m)}(s, t, 0) = \max_{e \in R(t^{(m)}, l)} [\delta^*(e, t^{(m)}) - \delta^*(e, s^{(m)})] \tag{8.4}$$

Note that the above upper bound ${U_\gamma}^{(m)}(s,t,0)$ is independent of the occurrence times of the events in the reference set of $s^{(m)}$. In other words, it does not depend on the history of the timed execution prior to segment $S^{(l)}$. Applying a longest path analysis from a fixed event e as outlined above, the term $\delta^*(e,t^{(m)})$ in (8.4) is computed in $O(|T(S)|+|P(S)|)$ time where $|P(S)|$ is the number of places in S. Thus, ${U_\gamma}^{(m)}(s,t,0)$ is computed in $O((|T(S)|+|P(S)|)*|R|)$ time where $|R|$ is the size of the referent set of e. This way, ${U_\gamma}^{(m)}(s,t,0)$ is computed in $O(|T(S)|+|P(S)|)$ time.

8.3 Bounding and evaluating TSE statistics

As pointed out earlier, the structure of segments are independent. However, multiple TSEs due to the same separation triple may start in one segment. Thus, these TSEs can be dependent on each other. To overcome this dependency, we treat all the TSE instances starting from one segment as a *group* and translate the problem of bounding the average TSE to that of bounding the average grouped TSE [88, 86]. In addition, extensions to variance and other higher order statistics of the TSE bounds can also be similarly proved. For the purposes of this work, we refer to the derived bound of any statistic (average, variance, etc..) of a TSE $\gamma(s,t,e)$ as $\mathcal{S}_\gamma(s,t,e)$. Evaluation of a statistic $\mathcal{S}$ can be performed via Monte-Carlo simulation (e.g.,[51]) in which each independent segment yields one sample of $\mathcal{S}_\gamma(s,t,e)$.

8.3.1 Grouping of TSEs

We know that the structures of segments are independent. However, multiple TSEs due to a same separation triple may start in one segment. These TSEs can be dependent. To overcome this dependency, we treat all the TSE instances starting from a same segment together as a *group* [88, 86].

Two TSEs $\gamma^{(i)}(s,t,\varepsilon)$ and $\gamma^{(j)}(s,t,\varepsilon)$ are belong to a group if $s^{(i)}$ and $s^{(j)}$ are in a same segment. The number of TSEs of the triple (s,t,ε) in group k is called the *length* of the group, denoted by $\alpha^{(k)}(s)$. If $\eta^{(k)}(s)$ is the index of the last TSE in group k, we have

$$\alpha^{(k)}(s) = \eta^{(k+1)}(s) - \eta^{(k)}(s).$$

Let us further define the sum of all TSEs in group k to be k-th *grouped TSE*, denoted by $\beta^{(k)}(s,t,\varepsilon)$. That is

$$\beta^{(k)}(s,t,\varepsilon) = \sum_{l=\eta^{(k-1)}(s)+1}^{\eta^{(k)}(s)} \gamma^{(l)}(s,t,\varepsilon).$$

It has been that both the group length sequence and the grouped TSE sequence are weakly ergodic [88]. In other words,

$$\frac{1}{n}\sum_{k=1}^{n} \beta^{(k)}(s,t,\varepsilon) \;\rightarrow\; \overline{\beta}(s,t,\varepsilon), \tag{8.5}$$

$$\frac{1}{n}\sum_{k=1}^{n} \alpha^{(k)}(s) \;\rightarrow\; \overline{\alpha}(s). \tag{8.6}$$

The convergence in (8.5) and (8.6) takes almost surely and in mean. Additionally,

$$\overline{\beta}(s,t,\varepsilon) = \overline{\alpha}(s)\overline{\gamma}(s,t,\varepsilon) \tag{8.7}$$

$$\overline{\alpha}(s) = \mathbf{E}\alpha^{(k)}(s) \tag{8.8}$$

Similarly, we define the grouped TSE-squares as the sum of squares of all the TSEs in a group. Similar to the argument of the existence of the TSE variance, we expect that the sequence of the grouped TSE-squares are also weakly ergodic.

8.3.2 Bounding TSE statistics

Following the argument in the previous section, the problem of bounding the average TSE is translated into bounding the average grouped TSE. Note that a grouped TSE is upper bounded by the sum of the upper bounds on the TSEs in the group. For a grouped TSE $\beta^{(k)}(s,t,0)$, we have

$$\beta^{(k)}(s,t,0) \leq U_{\beta}{}^{(k)}(s,t,0) = \sum_{m=\eta^{(k-1)}+1}^{\eta^{(k)}} U_{\gamma}{}^{(m)}(s,t,0) \tag{8.9}$$

Thus, the expectation $\mathbf{E}U_{\beta}{}^{(k)}(s,t,0)$ upper bounds the expectation $\mathbf{E}\beta^{(k)}(s,t,0)$, both taken over all possible timed executions. Because the grouped TSE sequence is weakly ergodic, $\mathbf{E}U_{\beta}{}^{(0)}(s,t,0)$ upper bounds $\overline{\beta}(s,t,0)$. In addition, extensions to variance and other higher order statistics of the TSE bounds can also be similarly proved [88, 86]

The following inequality gives the bounds on $\sigma^2(s,t,\varepsilon)$:

$$\lim_{n\to\infty}\frac{1}{n}\sum_{k=1}^{n}\min\{{U_\gamma}^{(k)^2}(s,t,\varepsilon),{L_\gamma}^{(k)^2}(s,t,\varepsilon)\}-\max\{\overline{U_\gamma}^2(s,t,\varepsilon),\overline{L_\gamma}^2(s,t,\varepsilon)\}\leq\sigma^2_{\gamma(s,t,\varepsilon)}\leq\lim_{n\to\infty}\frac{1}{n}\sum_{k=1}^{n}\max\{{U_\gamma}^{(k)^2}(s,t,\varepsilon),{L_\gamma}^{(k)^2}(s,t,\varepsilon)\}-\min\{\overline{U_\gamma}^2(s,t,\varepsilon),\overline{L_\gamma}^2(s,t,\varepsilon)\}$$

where $\overline{U_\gamma}(s,t,\varepsilon)$ and $\overline{L_\gamma}(s,t,\varepsilon)$ are the means of the upper and lower bounds on the TSE, respectively.

For the k-th TSE of the triple (s,t,ε), we have

$$\mathbf{1}_{{U_\gamma}^{(k)}(s,t,\varepsilon)\leq x}\leq\mathbf{1}_{\gamma^{(k)}(s,t,\varepsilon)\leq x}\leq\mathbf{1}_{{L_\gamma}^{(k)}(s,t,\varepsilon)\leq x} \tag{8.10}$$

where ${U_\gamma}^{(k)}(s,t,\varepsilon)$ and ${L_\gamma}^{(k)}(s,t,\varepsilon)$ are the upper and lower bounds on the TSE $\gamma^{(k)}(s,t,\varepsilon)$, respectively. The quantities $\mathbf{1}_{{U_\gamma}^{(k)}(s,t,\varepsilon)\leq x}$ and $\mathbf{1}_{{L_\gamma}^{(k)}(s,t,\varepsilon)\leq x}$ can be again determined using longest path analysis. Finally, using a technique similar to the one used in bounding the means of TSEs, we obtain bounds on $F_{\gamma(s,t,\varepsilon)}(x)$ for a fixed x. In practice, upper and lower bounds on $F_{\gamma(s,t,\varepsilon)}$ may be plotted at a sufficiently large number of points, say, 20, yielding a discretized approximation of the distribution.

8.3.3 Evaluating the bounds based on Monte-Carlo sampling

The previous section derives closed-form expressions of the bounds on various TSE statistics. These expressions take expectations of potentially complicated random variables. To evaluate these expressions to a given accuracy, the well-known Monte-Carlo approach (e.g.,[51]) has been adapted [88, 86].

Let us write W to denote the above random variable for which the expectation is taken. Let F_W be its distribution function.

The intuition behind Monte-Carlo approach is the Central Limit Theorem [32] which characterizes the partial sum of iid random variables. In our case, if $W_1, W_2, \cdots, W_n$ are iid random variables distribution F_W, the random variable $\frac{1}{n}S_n$ approaches $\mathbf{E}W$ as n grows, where $S_n = W_1 + W_2 + \cdots + W_n$. More precisely, $\frac{1}{n}S_n$ approaches the *normal* distribution with mean $\mathbf{E}W$ and variance $\frac{1}{n}\sigma_W^2$ where σ_W is the variance of W. Since $\frac{1}{n}\sigma_W^2$ decreases to 0 as n grows, any *realization* of the random variable $\frac{1}{n}S_n$ is a good (unbiased) estimate of $\mathbf{E}W$ when n is large. In fact, for a given *relative error interval* I_e and a confidence level L_c, $P\{|(\frac{1}{n}S_n - \mathbf{E}W)/\mathbf{E}W| < I_e\} > L_c$ if

$$n > (\frac{z_{x/2}\sigma_W}{I_e\mathbf{E}W})^2, \tag{8.11}$$

where $x = 1 - L_c$ and $z_{x/2}$ is defined such that the tail probability to its right under the standard normal distribution is $x/2$. Such a realization of $\frac{1}{n}S_n$ is known as *Monte-Carlo* sampling. It randomly realizes n iid variables (W_1 through W_n), yielding a *sample* of W of size n.

In practice, the variance of W, i.e., σ_W^2 is unknown and is commonly replaced by the variance of the sample, i.e., $S^2 = \frac{1}{n-1}\sum (W_i - \frac{1}{n}S_n)^2$. Similarly, $\mathbf{E}W$ is replaced by the mean of the sample, i.e., $\frac{1}{n}S_n$ itself. Consequently, $z_{x/2}$ in (8.11) with $x = 1 - L_c$ has to be replaced by $t_{x/2}$ which is defined such that the tail probability to its right under the t-distribution [51] is $x/2$.

In many cases, the upper and lower bounds on the TSE statistics derived in the previous subsections are very close to each other, which combined yield very good estimates. Nevertheless, the lower and upper bounds can be made even closer to each other by using some amount of *history* segments prior to the segments being considered. This effectively pushes the reference set further apart from the target TSE pairs [88, 86].

Strictly speaking, the applicability of Monte-Carlo approach to our analysis also depends on the finiteness of high order moments of the delays of the PTPNs. However, we do not believe this is a significant restriction because in real designs, delays of system components are either finite or their moments are finite up to a sufficiently high order.

Chapter 9

Reduced Petri net and its TSE statistics

One of the goals of this work is to take a PTPN N and reduce it to a smaller net N'. The key requirement is that N' should preserve all TSE statistics related to those signals also in N. We now introduce a class of reduced Petri nets for which we will prove that this performance-preservation property is guaranteed.

Definition 4 Reduced Petri Net: *A reduced Petri net* $N' = (P', T', F')$ *is a reduced Petri net of* $N = (P, T, F)$ *if,*

1. $T' \subseteq T$
2. $P'_c \equiv P_c$
3. $M'_0(p) = M_0(p)$, $\forall p \in P_c$
4. $(t, p) \in F \Leftrightarrow (t, p) \in F'$, $\forall p \in P_c \; \forall t \in T'$
5. $(p, t) \in F \Leftrightarrow (p, t) \in F'$, $\forall p \in P_c \; \forall t \in T'$

6. $\forall \psi' \in \Psi_{N'}(s,t)$, $\forall (s,t) \in (T',T')$, $\exists \psi \in \Psi_N(s,t)$ *such that* $(\Delta(\psi') = \Delta(\psi)) \wedge (M(\psi') = M(\psi))$

As an example, the reduced Petri net of the net in Figure 2.4(a) is depicted in 9.1(a).

To prove the performance-preservation property, we must introduce a few definitions. For a Petri net N and its reduced Petri net N', we call N_π and N'_π *corresponding event graphs* if the sequence of decisions is identical in N_π and N'_π, i.e., $p^k\bullet \in T_\pi = p^k\bullet \in T'_\pi$, $\forall p \in P_c$. A segment $S'(\xi', \theta')$ is a *corresponding segment* of the segment $S(\xi, \theta)$ if the partial order of S' is preserved in the partial order of S, i.e., $\theta' \subseteq \theta$.

Next, we need to define the following partial order operators.

Definition 5 *For a set of events E and an event e,*

$$e \preceq E \Leftrightarrow [(e \in E) \vee (\exists (e, e_i), e_i \in E)] \wedge [\neg(\exists (e_j, e), e_j \in E)]$$

$$e \succeq E \Leftrightarrow [(e \in E) \vee (\exists (e_i, e), e_i \in E)] \wedge [\neg(\exists (e, e_j), e_j \in E)]$$

$$e \prec E \Leftrightarrow (e \preceq E) \wedge (e \notin E)$$

$$e \succ E \Leftrightarrow (e \succeq E) \wedge (e \notin E)$$

Definition 6 *For a pair of sets of events E_1 and E_2,*

$$E_1 \preceq E_2 \Leftrightarrow (\forall e_i \in E_1, e_i \preceq E_2) \wedge (\forall e_j \in E_2, e_j \succeq E_1)$$

$$E_1 \succeq E_2 \equiv E_2 \preceq E_1$$

$$E_1 \prec E_2 \Leftrightarrow (E_1 \preceq E_2) \wedge (E_1 \cap E_2 = \emptyset)$$

$$E_1 \succ E_2 \equiv E_2 \prec E_1$$

Lemma 4 *For any sequence of segments which partitions N'_π, there exists a sequence of segments which partitions N_π such that the k-th segment in N'_π corresponds to the k-th segment in N_π where N' is the reduced Petri net of N, and N'_π is the corresponding event graph of N_π.*

Proof (Sketch) Let ξ'^k be the k-th cut and θ'^k be the partial order of the k-th segment where the k-th segment, $S'^k(\xi'^k, \theta'^k)$, is a segment between ξ'^k and ξ'^{k+1}. We have $\theta'^k = \{(s,t) | s \in T'^k \wedge t \in T'_\pi$ where T'^k is a set of events in a segment $S'^k\}$. Let $\bigcup_o^k$ and $\bigcup_o'^k$ be $\forall t \in \bigcup_{\forall p \in \xi^k} p\bullet$ and $\forall t \in \bigcup_{\forall p \in \xi'^k} p\bullet$ and $\bigcup_r^k$ and $\bigcup_r'^k$ be $\forall t \in \bigcup_{\forall p \in \xi^k} \bullet p$ and $\forall t \in \bigcup_{\forall p \in \xi'^k} \bullet p$. Figure 2.4(b) and 9.1(b) shows examples of $\bigcup_o^k$, $\bigcup_r^k$, $\bigcup_r'^k$ and $\bigcup_o'^k$.

For any ξ'^k of N'_π, there exists a ξ^k of N_π such that $\bigcup_r'^k \preceq \bigcup_r^k \prec \bigcup_o^k \preceq \bigcup_o'^k$ because N_π and N'_π are corresponding event graphs and $T'_\pi \subseteq T_\pi$. Thus, for any k-th segment of N'_π, there exists the k-th segment of N_π which yields a one-to-one mapping between segments in N'_π and N_π.

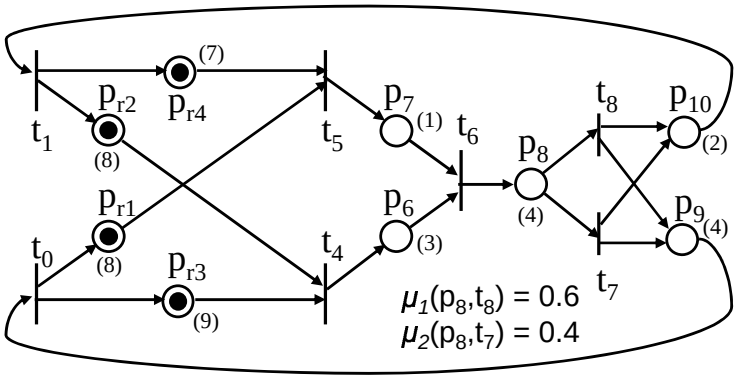
From Definition of the corresponding segments, if $\theta'^k \subseteq \theta^k$, then the one-to-one mapping maps corresponding segments. Since $\bigcup_o^k \preceq \bigcup_o'^k \preceq T'_k \preceq \bigcup_r'^{k+1} \preceq \bigcup_r^{k+1}$ and partial order operators are transitive in set-to-set operations, we have that $T'^k \subseteq T^k$. From Definition of the reduced Petri net, we have $\theta'_\pi \subseteq \theta_\pi$. $\theta'^k = \{(e_i, e_j)|e_i \in T'^k, e_j \in T'_\pi\}$ and $\theta^k = \{(e_i, e_j)|e_i \in T^k, e_j \in T_\pi\}$. Thus, $\theta'^k \subseteq \theta^k$ since $\theta'_\pi \subseteq \theta_\pi$, $T'^k \subseteq T^k$ and $T'_\pi \subseteq T_\pi$.

Lemma 5 *Let S'^k and S^k be the k-th segments of corresponding event graphs N'_π, N_π, respectively. Then, any upper (lower) bound $U'^{(k)}_\gamma(s,t,\epsilon)$ ($L'^{(k)}_\gamma(s,t,\epsilon)$) on a TSE instance of a pair of events (s,t) belonging to the segment S'^k is a bound on the TSE instance $\gamma^{(k)}(s,t,\epsilon)$ belonging to the segment S^k.*

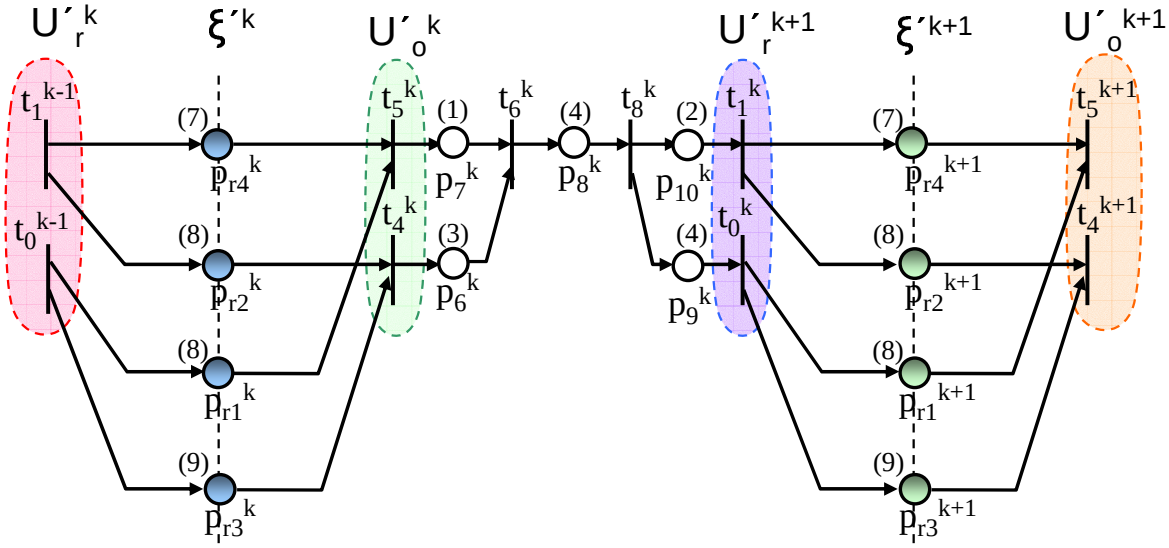
Proof (Sketch)

For any reference set, R'^k, of a target event t in S'^k, there exists a reference set, R^k, of the target event t in S^k such that $R^k \preceq R'^k$ because $\bigcup_o^k \preceq \bigcup_o'^k \preceq R'^k$. For example, we can choose $R^k = \bigcup_o^k$. Thus, for any events $r \in R^k$, we have

$$(r \in R^k) \Leftrightarrow (r \in R'^k) \vee \exists (r, r') \in \theta^k, r \in R, r' \in R'$$



(a) A reduced Petri net



(b) A timed execution of the Petri net in (a)

Figure 9.1: A reduced Petri net of the net in Figure 2.4(a) and its timed execution.

Case 1 $r \in R'^k$

$$\delta^*(r,t) = \delta'^*(r,t)$$

$$\delta^*(r,s) = \delta'^*(r,s)$$

$$\delta^*(r,t) - \delta^*(r,s) \leq U'^k(s,t) \tag{9.1}$$

Case 2 $\exists (r,r') \in \theta^k, r \in R, r' \in R'$

$$\delta^*(r,t) = \max_{r' \in R'^k_s} (\delta^*(r,r') + \delta^*(r',t)) \tag{9.2}$$

$$\delta^*(r,s) = \max_{r' \in R'^k_s} (\delta^*(r,r') + \delta^*(r',s)) \tag{9.3}$$

Let r'_j and r'_i be such that,

$$\delta^*(r,t) = \delta^*(r,r'_j) + \delta^*(r'_j,t) \tag{9.4}$$

$$\geq \delta^*(r,r'_i) + \delta^*(r'_i,t) \tag{9.5}$$

$$\delta^*(r,s) = \delta^*(r,r'_i) + \delta^*(r'_i,s) \tag{9.6}$$

$$\geq \delta^*(r,r'_j) + \delta^*(r'_j,s) \tag{9.7}$$

From equations (9.4) and (9.7)

$$\begin{aligned}\delta^*(r,t) - \delta^*(r,s) &\leq \delta^*(r,r'_j) + \delta^*(r'_j,t) - \delta^*(r,r'_j) - \delta^*(r'_j,s) \\ &\leq \delta^*(r'_j,t) - \delta^*(r'_j,s) \end{aligned} \tag{9.8}$$

From (9.8),

$$\delta^*(r,t) - \delta^*(r,s) \leq \delta^*(r'_j,t) - \delta^*(r'_j,s) \leq U'^k(s,t) \tag{9.9}$$

Since the above equation is true for every $r \in R^k$, $U^k(s,t) \leq U'^k(s,t)$. Similarly, the following inequality, $L^k(s,t) \geq L'^k(s,t)$, can easily be proved through the duality of bounds. Thus, any bound on a TSE instance of a pair of events belonging to the segment S'^k is a bound on the TSE instance belonging to the segment S^k.

Theorem 7 *Let N' be a reduced Petri net of N. Then, any bound of a TSE statistic in N' defined by $\mathcal{S}'(s,t,e)$ is a bound of the corresponding TSE statistic in N.*

Proof (Sketch) From Lemma 4, there exists a sequence of segments which partitions N_π such that k-th segment in N'_π is a corresponding k-th segment in N_π. From extensions of Lemma 5 to grouped TSEs, we know that each sample of $\mathcal{S}'(s,t,e)$ is also a bound for the corresponding TSE statistic in N. Thus, the resulting

probabilistic bound obtained by the Monte-Carlo simulation for $\mathcal{S}'(s,t,e)$ is also valid for N.[1]

[1]Note, however, that the obtained bounds of TSE statistics of N_π and N'_π may still be numerically different due to differences in cut and/or the reference set selection.

Chapter 10

Petri net Reduction Operations

In this section, we propose two reduction operations and prove that the modified Petri net through those operations is the reduced Petri net of the original net.

10.1 Projection

Reduction Rule 1.(Projection)

Precondition: There exist two sets of transitions T_i, T_o, two sets of places P_i, P_o, and a transition t_d.

1. $|\bullet p| = |p\bullet| = 1$, $\forall p \in P_i$ and P_o.

2. $(s, p) \in F$ and $(p, t_d) \in F$ where $s \in T_i$, $\forall p \in P_i$

3. $(t_d, p) \in F$ and $(p, t) \in F$ where $t \in T_o$, $\forall p \in P_o$

Rule: For every pair of places (p_i, p_o), $p_i \in P_i, p_o \in P_o$, create new place $p_n \in P_n$ such that $d(p_n) = d(p_i) + d(p_o)$, $M(p_n) = M(p_i) + M(p_o)$, $\bullet p_n = \bullet p_i$ and $p_n\bullet = p_o\bullet$.

Remove $p_i \in P_i, p_o \in P_o$ and t_d.

After applying projection operation to the Petri net N, the modified Petri net N' is a reduced Petri net of N. Since only transition t_d is removed from N, $T' \subseteq T$ and the places with choice are not considered in this operation, hence properties 1-5 of the reduced Petri net are satisfied. Furthermore, for any pair of transitions (t_i, t_o) where $t_i \in T_i$ and $t_o \in T_o$, there exists a path from t_i to t_o with equal path delay and number of initial markings. Thus all properties of the reduced Petri net are satisfied. Figure 10.1 illustrates an example of the projection operation.

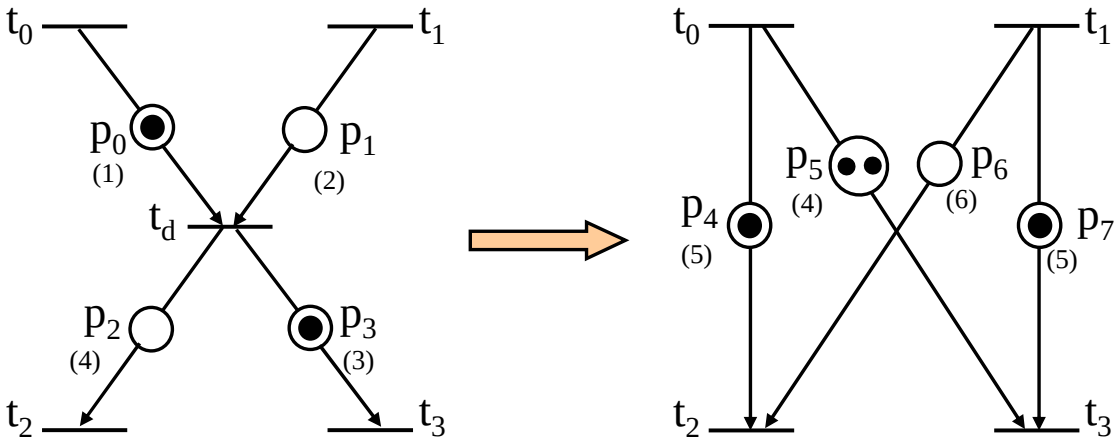


Figure 10.1: Projection of t_d

10.2 Redundancy removal

Reduction Rule 2.(Redundancy Removal)

Precondition: There exist a transitions t_i and a transition t_o and a place p_d.

1. $\bullet p_d = \{t_i\}$ and $p_d \bullet = \{t_o\}$

2. $\exists \psi \in \Psi(t_i, t_o)$ such that $p_d \notin \psi$, $\Delta(\psi) \geq d(p_d)$ and $M(\psi) = M(p_d)$.

Rule: Remove p_d.

After applying redundancy removal operation to the Petri net N, a modified Petri net N' is a reduced Petri net of N. Since all transitions remains the same and the places with choice are not considered in this operation, properties 1-5 of the reduced Petri net are satisfied. Furthermore, for a pair of transitions (t_i, t_o), there exists a path from t_i to t_o with equal path delay and number of initial markings. Thus, all properties of the reduced Petri net are satisfied. Figure 10.2 illustrates an example of redundancy removal operation.

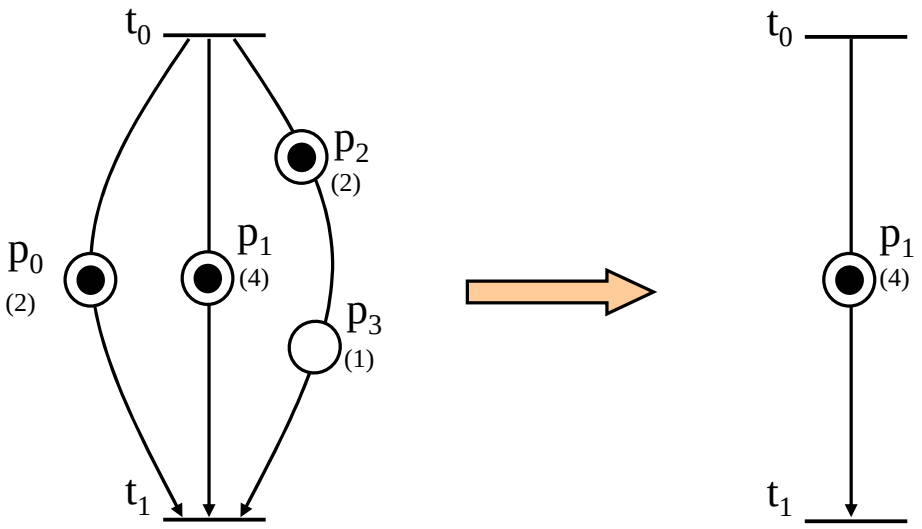


Figure 10.2: An example of redundancy removal

10.3 Case Study

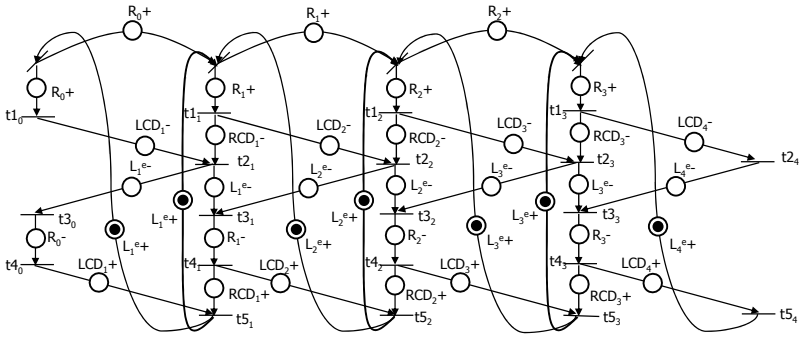
We now demonstrate the power of the proposed reduction operations on a well-known 4-phase Petri net model of a 3-stage PCHB linear pipeline illustrated in Figure 10.3. To semi-automate reduction, we developed a C program that would check the validity of user-proposed projection and redundancy removal reductions and if valid return the reduced Petri net .

Table 1 shows the number of places/transitions in the input model and the percentage of places/transitions that were removed compared to the original model due to the proposed reductions. As shown in Table 1, after the proposed series of reductions, the number of places were reduced by more than half and the number of transitions were reduced by a factor of 6. Thus, using this model in place of its more complicated orignal model for system-level performance analysis would yield a significant run-time improvement.

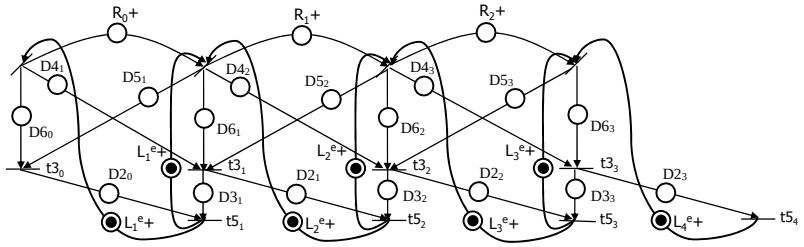
In addition, the reduced Petri net highlights performance characteristics and dependencies not obvious in the original, more complicated, net. For example, R_i is the typical forward latency, $B1_i$ is the typical reverse latency, and $B2_i$ is a special reverse latency typical of pipelines based on half-buffer templates. $B2_i$ highlights the dependency across three pipeline stages that was not obvious from the more detailed Petri nets or, for that matter, the original description of PCHB in [46].

	# places	reduc_p	# transitions	reduc_t
PCHB model	37	0%	24	0%
Intermediate I	26	35%	12	50%
Intermediate II	22	41%	8	67%
Reduced model	15	59%	4	83%

Table 10.1: An Example of Petri net reduction on 3-stage PCHB model.



(a) Initial PCHB model



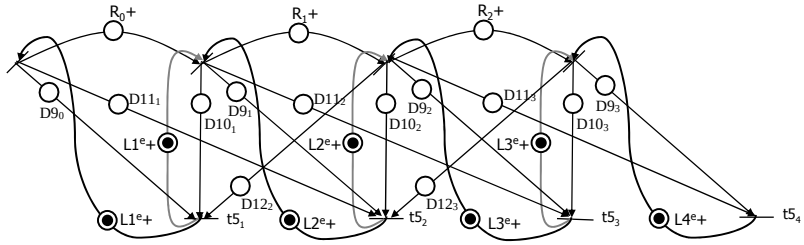
Proj(t4$_i$): $D2_i = R_i\text{-} + LCD_{i+1}+$; $D3_i = R_i\text{-} + RCD_i+$

Proj(t1$_i$): $D0_i = R_i+ + LCD_{i+1}\text{-}$; $D1_i = R_i+ + RCD_i\text{-}$;

Proj(t2$_i$): $D4_i, D6_i = D0_i + L_i^e\text{-}$; $D5_i, D7_i = D1_i+ + L_i^e\text{-}$;

RR(D7$_i$): $D6_{i+1} = \max(D7_i, D6_{i+1})$

(b) Intermediate model I

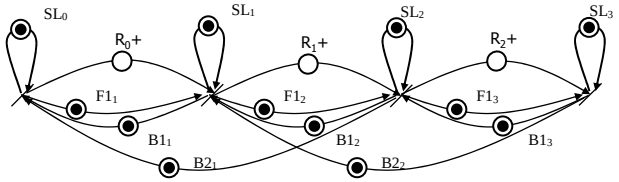


Proj(t3$_i$): $D9_i = D8_i + D2_i$; $D14_i = D4_i + D3_i$; $D10_i = D8_i + D3_i$; $D13_i = D5_{i+1} + D2_i$
$D11_i = D4_i + D2_i$; $D12_i = D5_i + D3_{i-1}$

RR(D14$_{i+1}$): $D9_i = \max(D9_i, D14_{i+1})$

RR(D13$_{i-1}$): $D10_i = \max(D10_i, D13_{i-1})$

(c) Intermediate model II



RR(L1^e+): Remove all gray edges since $L_i^e+ \leq R_{i-1} + L_i^e+$

Proj(t3$_i$): $F1_i = D11_i + L_{i+1}^e+$; $B1_i = D10_i + L_i^e+$; $B2_i = D12_{i+1} + L_i^e+$; $SL_i = D9_i + L_{i+1}^e+$;

(d) Reduced PCHB model

Figure 10.3: Reduction of 3-stage PCHB pipeline model.

Chapter 11

Conclusion and Future work

In this work, we characterizes a class of Petri net reduction operations that preserve TSE statistics as well as two useful operators that fall into this class. We applied these two operations using a novel semi-automated Petri net reduction tool to a Petri net model of well-known asynchronous pipeline. The resulting reduced Petri net is both significantly and adds insight into its performance characteristics. Our tool can be used to obtain performance models for libraries of asynchronous cells on which system-level performance analysis tools can be applied. Potential future work includes the development of more reduction operations, the further automation of the tool, and the expansion of our work to more general stochastic Petri nets.

Reference List

[1] V. Akella and G. Gopalakrishnan. SHILPA: A high-level synthesis system for self-timed circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 587–591. IEEE Computer Society Press, November 1992.

[2] Asynchronous Digital Design, Inc. Asynchronous Digital Design Homepage. `http://www.avlsi.com/`.

[3] Brandon M. Bachman, Hao Zheng, and Chris J. Myers. Architectural synthesis of timed asynchronous systems. In *Proc. International Conf. Computer Design (ICCD)*. IEEE Computer Society Press, October 1999.

[4] Rosa M. Badia and Jordi Cortadella. High-level synthesis of asynchronous systems: Scheduling and process synchronization. In *Proc. European Conference on Design Automation (EDAC)*, pages 70–74. IEEE Computer Society Press, February 1993.

[5] Peter A. Beerel. *CAD Tools for the Synthesis, Verification, and Testability of Robust Asynchronous Circuits*. PhD thesis, Stanford University, 1994.

[6] Peter A. Beerel. Asynchronous circuits: An increasingly practical design solution. In *Proc. of the International Symposium on Quality Electronic Design (ISQED)*, 2002.

[7] Peter A. Beerel, Sangyun Kim, Pei-Chuan Yeh, and Kyeounsoo Kim. Statistically optimized asynchronous barrel shifters for variable length codecs. In *International Symposium on Low Power Electronics and Design*, pages 261–263, August 1999.

[8] M. Benes, S. M. Nowick, and A. Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 43–56, 1998.

[9] Kees van Berkel and Arjan Bink. Single-track handshaking signaling with application to micropipelines and handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 122–133. IEEE Computer Society Press, March 1996.

[10] Kees van Berkel, Ferry Huberts, and Ad Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Asynchronous Design Methodologies*, pages 99–106. IEEE Computer Society Press, May 1995.

[11] E. Best. Partial order behavior and structure of Petri nets. *Formal Aspects of Computing*, 2:123–138, 1990.

[12] E. Best and K. Voss. Free choice systems have home states. *Acta Informatica*, 21:89–100, 1984.

[13] Erik Brunvand. Parts-R-Us: A chip aparts... Technical Report CMU-CS-87-119, Carnegie Mellon University, May 1987.

[14] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[15] J. Campos, G. Chiola, J. M. Colom, and M. Silva. Properties and performance bounds for timed marked graphs. *IEEE Transactions on Circuits and Systems–I: Fundamental theory and applications*, 39(5):386–401, May 1992.

[16] Supratik Chakraborty, David L. Dill, and Kenneth Y. Yun. Min-max timing analysis and an application to asynchronous circuits. *Proceedings of the IEEE*, 87(2):332–346, February 1999.

[17] Wei-Chun Chou, Peter A. Beerel, and Kenneth Y. Yun. Average-case technology mapping of asynchronous burst-mode circuits. *IEEE Transactions on Computer-Aided Design*, 18(10):1418–1434, October 1999.

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

[19] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor, and Alexandre Yakovlev. Decomposition and technology mapping of speed-independent circuits using Boolean relations. *IEEE Transactions on Computer-Aided Design*, 18(9), September 1999.

[20] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

[21] Uri Cummings, Andrew Lines, and Alain Martin. An asynchronous pipelined lattice structure filter. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 126–133, November 1994.

[22] Jo Ebergen. Squaring the FIFO in GasP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 194–205. IEEE Computer Society Press, March 2001.

[23] Jo Ebergen and Robert Berks. Response time properties of linear asynchronous pipelines. *Proceedings of the IEEE*, 87(2):308–318, February 1999.

[24] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1987.

[25] Karl M. Fant and Scott A. Brandt. Null covention logic$_t m$. Technical report, Theseus Logic Inc. 2177 Youngman Ave., ST. Paul, MN, US, 1997.

[26] Marcos Ferretti and Peter A. Beerel. Single-track asynchronous pipeline templates using 1-of-N encoding. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1008–1015, March 2002.

[27] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. AMULET1: A micropipelined ARM. In *Proceedings IEEE Computer Conference (COMPCON)*, pages 476–485, March 1994.

[28] S. B. Furber, D. A. Edwards, and J. D. Garside. AMULET3: a 100 MIPS asynchronous embedded processor. In *Proc. International Conf. Computer Design (ICCD)*, September 2000.

[29] S. B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[30] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[31] J. D. Garside, S. B. Furber, and S.-H. Chung. AMULET3 revealed. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 51–59, April 1999.

[32] G. R. Grimmett and D. R. Stirzaker. *Probability and Random Processes (2nd Edition)*. Oxford Science Publications, 1992.

[33] Gary D. Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[34] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[35] M. A. Holliday and M. Y. Vernon. A generalized timed Petri net model for performance analysis. *IEEE Transactions on Software Engineering*, 13(12):1297–1310, December 1987.

[36] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. *IEEE Transactions on Computers*, 44(11):1306–1317, November 1995.

[37] Henrik Hulgaard. *Timing Analysis and Verification of Timed Asynchronous Circuits*. PhD thesis, Department of Computer Science, University of Washington, 1995.

[38] Henrik Hulgaard and Steven M. Burns. Bounded delay timing analysis of a class of CSP programs with choice. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11, November 1994.

[39] Sung Tae Jung and Chris J. Myers. Direct synthesis of timed asynchronous circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 332–337, November 1999.

[40] R. M. Karp. A characterization of the minimum cycle mean in a diagraph. *Discrete mathematics*, 23:309–311, 1978.

[41] Joep Kessels and Paul Marston. Designing asynchronous standby circuits for a low-power pager. *Proceedings of the IEEE*, 87(2):257–267, February 1999.

[42] Hoshik Kim. Relative timing based verification of timed circuits and systems. In *Proc. International Workshop on Logic Synthesis*, June 1999.

[43] Hoshik Kim, Peter A. Beerel, and Ken Stevens. Relative timing based verification of timed circuits and systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 115–124, April 2002.

[44] C. E. Leiserson and J. B. Saxe. Optimizing synchronous systems. *Journal of VLSI Computer System*, 1:41–67, 1983.

[45] Michiel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, and Alex Kondratyev. Asynchronous design using commercial HDL synthesis tools. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 114–125. IEEE Computer Society Press, April 2000.

[46] Andrew M. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1996.

[47] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[48] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[49] Alain J. Martin. Synthesis of asynchronous VLSI circuits. In J. Straunstrup, editor, *Formal Methods for VLSI Design*, chapter 6, pages 237–283. North-Holland, 1990.

[50] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Pénzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997.

[51] I. R. Miller, J. E. Freund, and R. Johnson. *Probability and Statistics for Engineers*. Prentice Hall, 1990.

[52] M. K. Molloy. *On the Integration of delay and throughput measures in distributed processing models*. PhD thesis, University of California, Los Angeles, 1981.

[53] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77:541–580, April 1989.

[54] Chris J. Myers, Wendy Belluomini, Kip Killpack, Eric Mercer, Eric Peskin, and Hao Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 335–340, February 2001.

[55] Chris J. Myers and Teresa H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[56] T. Nanya, A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, F. Okamoto, H. Fujimoto, O. Fujita, M. Yamashina, and M. Fukuma. TITAC-2: A 32-bit scalable-delay-insensitive microprocessor. In *Symposium Record of HOT Chips IX*, pages 19–32, August 1997.

[57] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, 1994.

[58] Steven M. Nowick, Kenneth Y. Yun, and Peter A. Beerel. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 210–223. IEEE Computer Society Press, April 1997.

[59] Mika Nyström. *Asynchronous Pulse Logic*. PhD thesis, California Institute of Technology, May 2001. Caltech Computer Science Technical Report 2001.011.

[60] Mika Nyström and Alain Martin. *Asynchronous Pulse Logic*. Kluwer Academic Publishers, 2002.

[61] Recep O. Ozdag and Peter A. Beerel. High-speed QDI asynchronous pipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 13–22, April 2002.

[62] Recep O. Ozdag, Montek Singh, Peter A. Beerel, and Steven M. Nowick. High-speed non-linear asynchronous pipelines. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1000–10007, March 2002.

[63] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Prentice Hall, 1995.

[64] Ad M. G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996.

[65] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on Software Engineering*, 6(5):440–449, September 1980.

[66] Shai Rotem, Ken Stevens, Ran Ginosar, Peter Beerel, Chris Myers, Kenneth Yun, Rakefet Kol, Charles Dike, Marly Roncken, and Boris Agapiev. RAPPID: An asynchronous instruction length decoder. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70, April 1999.

[67] Sadiq M. Sait and Habib Youssef. *VLSI Physical Deign Automation*. IEEE Press, 1995.

[68] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[69] N. Shenoy. Retiming: Theory and practice. *Integration, the VLSI journal*, 22:1–21, 1997.

[70] M. Silva and J. Campos. Structural performance analysis of stochastic Petri nets. In *IEEE International Computer Performance and Dependability Symposium*, pages 61–70, 1995.

[71] Montek Singh and Steven M. Nowick. Fine-grain pipelined asynchronous adders for high-speed DSP applications. In *Proceedings of the IEEE Computer Society Workshop on VLSI*, pages 111–118. IEEE Computer Society Press, April 2000.

[72] Montek Singh and Steven M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 198–209. IEEE Computer Society Press, April 2000.

[73] Fabio Somenzi. *Private Communications*, 1999. F. Somenzi is a professor of computer science at the University of Colorado.

[74] Jens Sparsø and Jørgen Staunstrup. Delay-insensitive multi-ring structures. *Integration, the VLSI journal*, 15(3):313–340, October 1993.

[75] Ken Stevens, Ran Ginosar, and Shai Rotem. Relative timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, April 1999.

[76] Ivan Sutherland and Scott Fairbanks. GasP: A minimal FIFO control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53. IEEE Computer Society Press, March 2001.

[77] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[78] Hiroaki Terada, Souichi Miyata, and Makoto Iwata. DDMP's: Self-timed super-pipelined data-driven multimedia processors. *Proceedings of the IEEE*, 87(2):282–296, February 1999.

[79] M. Theobald and S. M. Nowick. An implicit method for hazard-free two-level minimization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 58–69, 1998.

[80] Theseus Logic, Inc. Theseus Logic Homepage. `http://www.theseus.com/`.

[81] Sunan Tugsinavisut and Peter A. Beerel. Control circuit templates for asynchronous bundled-data pipelines. In *Proc. Design, Automation and Test in Europe (DATE)*, page 1098, March 2002.

[82] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.

[83] Jiacun Wang. *Timed Petri Nets*. Kluwer Academic Publishers, 1998.

[84] Ted E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, June 1991.

[85] A. Xie and P. A. Beerel. Efficient state classification of finite state Markov chains. *IEEE Transactions on Computer-Aided Design*, 17(12):1334–1338, December 1998.

[86] A. Xie and P. A. Beerel. Performance analysis of asynchronous circuits and systems using stochastic timed Petri nets. In A. Yakovlev, L. Gomes, and L. Lavagno, editors, *Hardware Design and Petri Nets*, pages 239–268. Kluwer Academic Publishers, March 2000.

[87] Aiguo Xie and Peter A. Beerel. Symbolic techniques for performance analysis of timed systems based on average time separation of events. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 64–75. IEEE Computer Society Press, April 1997.

[88] Aiguo Xie, Sangyun Kim, and Peter A. Beerel. Bounding average time separations of events in stochastic timed Petri nets with choice. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 94–107, April 1999.

[89] A. Yakovlev and A. M. Koelmans. Petri nets and digital hardward design. In *Lectures on Petri nets II: Basic Models*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[90] Kenneth Y. Yun, Peter A. Beerel, Vida Vakilotojar, Ayoob E. Dooply, and Julio Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. *IEEE Transactions on VLSI Systems*, 6(4):643–655, December 1998.