SEEKING EQUILIBRIUM BETWEEN COMMUNICATION AND COMPUTATION IN SYSTEM-LEVEL DESIGN

by

Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M03/24

5 May 2003

SEEKING EQUILIBRIUM BETWEEN COMMUNICATION AND COMPUTATION IN SYSTEM-LEVEL-DESIGN

by

Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M03/24

5 May 2003

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Seeking Equilibrium between Communication and Computation in System-Level Design

Luca P. Carloni   Alberto L. Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, Berkeley, CA 94720-1772
{lcarloni,alberto}@ic.eecs.berkeley.edu

May 5, 2003

Abstract

Designers who integrate various Intellectual Property (IP) cores in modern system-on-chips (SOC) face the important challenges of balancing the differences in throughput and latency between the IPs. This task is made more critical by the increasing role played by on-chip communication, as wire delays become dominant in deep sub-micron technologies and SOCs effectively behave like distributed systems. We argue that SOC design will be an average-case design, driven by global throughput metrics, as opposed to a worst-case design, driven by the minimization of local critical-path timing violations. We present a conceptual model for the development of a new class of design enviroments aimed to assist designers in the quest for a balanced design. The proposed approach guides the designers in abstracting away the functional details of each component to focus instead on tuning the values of those properties that impact the performance of the overall system. Synchronization, concurrency, and latency/throughput trade-offs become first-class citizens within a design framework that targets the derivation of a balanced computational architecture by exploiting parallelism and pipelining at a fine level of granularity. Besides the conceptual model, the main contributions of the paper include the definition of the throughput equalization problem and its solution by means of an exact algorithm.

1 Introduction

By offering millions of wirable gates with several levels of interconnect, state-of-the-art deep sub-micron technologies enable the realization of products with a broad diversity of applications as a single system-on-chip (SOC) with higher performance and lower cost. Intellectual Property (IP) reuse is considered a key technique to manage the increasing complexity of SOC design: designers are provided access to a library of large pre-designed modules, or *cores*, with the goal of enabling them to rapidly realize their systems by simply composing these cores. However, the large variety of options available make this task difficult, while short schedules rush early critical decisions that are quite costly to change later on in the design process [4].

IP core selection is based mainly on functionality requirements and ball-park considerations on the system performance that designers would like to obtain with the given technology process. Alternative IP cores with the same functionality present different figures in terms of throughput and latencies and, if selected, will impact differently the performance of the overall system. Selecting the right components for both computation and communication and assembling them so that they are synchronized and operate with a good level of concurrency is a task that may turn out quite challenging and that is definitely time-consuming. In fact, the important decision of choosing a certain component implies immediately a sequence of actions to integrate it on the chip and properly interface it with the rest of the system, with obvious consequences on the design time. After a series of several IP selections, interleaved by hours of subsequent simulation and design, the major risk is to end up focusing on local timing-closure issues without improving the global system performance. In the end, strict time-to-market constraints may preclude designers from reverting decisions and making a major global design re-organization, thereby leaving them with a sub-optimal design, or an unbalanced-design, or both. Furthermore, as discussed in [4], most designers rely on a register-transfer-level (RTL) system specification and design flow for both implementation and analysis. Instead, to address the SOC challenges requires a design methodology with the ability to evaluate options and make critical architectural decisions based on a system-level representation in advance of an RTL design.

This paper provides the means for performance engineering in SOC design. So far, performance and functionality have been intertwined in the ASIC flow that is centered around RTL synthesizable hardware-description language (HDL) specifications. This paper for the first time offers a capability which has been long available within a clock cycle, by means of various combinational logic speed-up techniques, but which has been prohibitively complex to manage at the sequential logic level. Traditional Extended Finite State Machine-based modeling paradigms break down when they need to confront a delay of one more or one less clock cycles in one of the components of a complex circuit. Asynchronous techniques, which would be best to manage the problem, are too far from mainstream use to be applicable in an ASIC design flow. Yet, any designer knows that when his latency is about 20 ns, he can split it into 4 or 5 pipeline stages, this achieving either 200 or 250MHz operation. The problem is that no design technique today supports him in this task.

To address these issues we present an interactive design framework that automatically analyzes the impact of the design decisions on the performance of the global system and synthesizes alternative better solutions ranked in order of optimality. Our throughput equalization algorithm provides designers with a tool to tackle the problem globally, with means other than simulation and intuition. Our approach is interactive, allows consideration of performance early in the design cycle, and targets the achievement of a balanced design that will enhance the average-case performance of the system.

Software designers have long known that functionality and performance are orthogonal requirements, that can be satisfied independent of each other, by optimizing small modules, and using a model of computation, e.g. Kahn Process or dataflow networks, that ensures the independence of computed results on the performance of each module. In this paper, we use Marked Graphs as a formal model to capture both the synchronization requirements among system components, and the flexibility in trading off cycle time and latency. This works can be seen as a generalization of the well-known concept of retiming [8, 13], since it includes the performance aspect of interconnection into the retiming graph, and it allows us to optimize *simultaneously* the performance impact of more or less pipelining and of more or less communication latency. As the final chip implementation is derived, our techniques can be combined with the result of recent works on throughput-driven on-chip communication synthesis [9, 12]. Previously, even high-level synthesis approaches that considered latency as a parameter, to optimize area and throughput, were looking only at one process at a time. In this work for the first time we consider multiple processes by using a global cost function, while [7] looked at process interaction locally, in a greedy fashion. On the other hand, [10] considered only rate analysis, rather than using rates for efficient synthesis and optimization. Finally, the theory of latency-insensitive design [3] does provide an automatic way to build interface logic around IP cores that make them robust to arbitrary latency-variations, but it has been focusing mainly on the *a posteriori* correction of latency communication mismatches (due to long interconnect delays) without considering the opportunity of automatically re-organizing the balance between communication and computation.

After providing some basic background on Petri Nets and Marked Graphs in Section 2, in Section 3 we discuss our adoption of Marked Graphs (MG) as the underlying formal model for the system exploration effort. We use MGs as the common semantic abstraction to model both the stand-alone instances from the IP core libraries as well as the systems that are obtained by composing them. The convenience offered by MGs to formally capture synchronization as well as performance properties (throughput, latency) allows us to define the throughput equalization problem as a vehicle to guide the system-level design exploration. In Section 5, we present an algorithm to solve exactly this problem and we illustrate it by means of simple examples. Finally, in

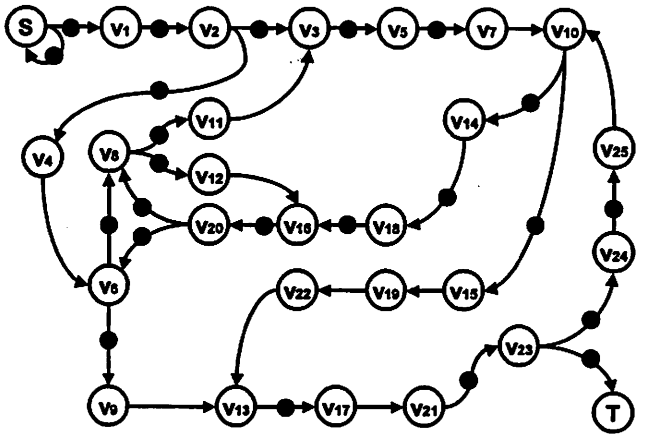

Figure 1: A Marked Graph with $\tau_{min}(\mathcal{MG}) = \frac{10}{5} = 2$.

Section 6 we draw some concluding remarks.

2 Background: Marked Graphs

We provide here just a basic set of definitions that are necessary for the rest of the paper, while we refer to the work by T. Murata [11] for a complete presentation of both Petri Nets and Marked Graphs.

Definition 2.1 *A* Petri Net *is a 5-tuple, $\mathcal{PN} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{W}, \mathcal{M}_0)$ where $\mathcal{P}$ is a finite set of places, $\mathcal{T}$ is a finite set of transitions, $\mathcal{F} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ is a set of arcs (flow relation), $\mathcal{W} : \mathcal{F} \to \mathbf{Z}^+$ is a weight function, $\mathcal{M}_0 : \mathcal{P} \to \mathbf{Z}^*$ is the initial marking, and such that $\mathcal{P} \cap \mathcal{T} = \emptyset \wedge \mathcal{P} \cup \mathcal{T} = \emptyset$. A Petri Net is said to be* ordinary *if all of its arc weights are equal to one. A Petri Net is said to be* safe *if the number of tokens in each place is not greater than one for any marking reachable from $\mathcal{M}_0$.*

In order to simulate the dynamic behavior of a system a marking in a Petri Net is changed according to the following *firing rules*:

1. A transition $t \in \mathcal{T}$ is said to be *enabled* if each input place p of t is marked with at least $w(p,t)$ tokens, where $w(p,t)$ is the weight of the arc from p to t.
2. An enabled transition may or may not fire (depending on whether or not the event takes actually place).
3. A firing of an enabled transition t removes $w(p,t)$ tokens from each input place p of t, and adds $w(t,p)$ tokens to each output place p of t.

A Petri Net is said to be *live* for the initial marking $\mathcal{M}_0$ if, no matter what marking has been reached from $\mathcal{M}_0$, it is possible to ultimately fire *any* transition of the net by progressing through some further firing sequence. A Petri Net that is not *live* is *deadlocked.*

Definition 2.2 *A* Marked Graph *$\mathcal{MG} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{W}, \mathcal{M}_0)$ is an ordinary Petri Net such that each place $p \in \mathcal{P}$ has exactly one input transition and exactly one output transition. A Marked graph $\mathcal{MG}$ is* consistent *if there*

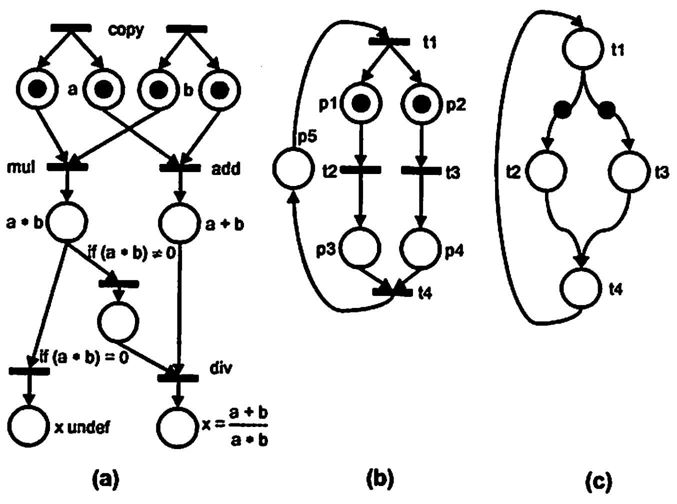

Figure 2: Petri Net and Marked Graph Examples.

exists a marking $\mathcal{M}_0$ and a firing sequence σ from $\mathcal{M}_0$ back to $\mathcal{M}_0$ such that every transitions occurs at least once in σ.

Figure 2 illustrates the above definitions through some examples adapted from [11]: Figure 2(a) shows a Petri Net representing a dataflow that computes the expression $x = \frac{a+b}{a-b}$. Notice that this is not a marked graph because the place labeled "$a-b$" has two output transitions. Figure 2(b) shows a marked graph representing parallel activities in a deterministic system: transitions t_2 and t_3 may occur in parallel, while they are always preceded by one occurrence of transition t_1 and followed by one occurrence of transition t_4. Since each place in a marked graph has exactly one incoming arc and exactly one outgoing arc (both with unit weight), marked graphs can be drawn more simply as *marked directed graphs*, where arcs correspond to places, vertices to transitions, and tokens are placed on arcs. This is the representation that we use in the rest of the paper. The marked graph of Figure 2(b) can be redrawn as the marked directed graph shown in Figure 2(c).

Definition 2.3 *A Marked Graph $\mathcal{MG}$ is* timed *if there exists a delay t_i associated with each transition in $t_i \in \mathcal{T}$. The* cycle time *τ of a consistent timed $\mathcal{MG}$ is the time to complete a firing sequence σ leading back to the starting marking. The* minimum cycle time *τ_{min} of a marked graph $\mathcal{MG}$ can be computed exactly as:*

$$\tau_{min}(\mathcal{MG}) = \max_{c \in C(\mathcal{MG})} \left(\frac{\sum_{t_i \in c} d(t_i)}{\mathcal{M}_0(c)} \right) \qquad (1)$$

where $C(\mathcal{MG})$ denotes the set of cycles in $\mathcal{MG}$ and $\mathcal{M}_0(c)$ denotes the number of tokens in cycle $c \in C(\mathcal{MG})$ at $\mathcal{M}_0$. A cycle c whose cycle time coincides with $\tau_{min}(\mathcal{MG})$ is said critical.

Figure 1 illustrates a marked graph $\mathcal{MG}$ having 6 cycles. These are reported in Table 1 together with the corresponding cycle time that have been computed assuming that $\forall t_i \in \mathcal{T}(\mathcal{MG})$, $d(t_i) = 1$. Hence, $\mathcal{MG}$ has minimum cycle time $\tau_{min}(\mathcal{MG}) = \frac{10}{5}$.

By noticing that the firing of a transition removes one token from each of the incoming arcs and it adds one token to each of the outgoing arcs, it is easy to see that the number of tokens in any cycle of a marked graph is constant. Also, for a connected marked graph with initial marking $\mathcal{M}_0$, a firing sequence σ can lead back to

c_i	vertices (transitions)	$\mathcal{M}_0(c)$	$\tau(c_i)$
c_1	$\{v_{10}, v_{15}, v_{19}, v_{22}, v_{13}, v_{17}, v_{21}, v_{23}, v_{24}, v_{25}\}$	5	10/5
c_2	$\{v_8, v_{12}, v_{16}, v_{20}\}$	3	4/3
c_3	$\{v_{10}, v_{14}, v_{18}, v_{16}, v_{20}, v_6, v_9, v_{13}, v_{17}, v_{21}, v_{23}, v_{24}, v_{25}\}$	10	13/10
c_4	$\{v_{10}, v_{14}, v_{18}, v_{16}, v_{20}, v_8, v_{11}, v_3, v_5, v_7\}$	8	10/8
c_5	$\{v_6, v_8, v_{12}, v_{16}, v_{20}\}$	4	5/4
c_6	$\{v_{10}, v_{14}, v_{18}, v_{16}, v_{20}, v_6, v_8, v_{11}, v_3, v_5, v_7\}$	9	11/9

Table 1: Cycles and cycle times for the marked graph of Figure 1.

$\mathcal{M}_0$ if and only if it fires every transition an equal number of times. Formal proves of these facts can be found in [1].

3 The Conceptual Model

The conceptual model presented in this section is the underpinning of our approach. We use marked graphs as the common semantic abstraction to model both the communication and computation features of the single components of a given IP library as well as the interaction of the components instanced into the design and the performance of the overall system. Marked graphs are good to model deterministic (decision-free) concurrent systems (and their synchronization mechanisms) and, among the various models of computation that are derivative of Petri Nets, they are considered the most amenable to analysis [11]. As we anticipated above, we want to abstract away the details of the format and the type of the data that are computed by the modules and are transmitted on the channels to simply focus on the presence/absence of data elements on the channels. Hence, we use the notion of *token* to denote the presence of a new data item on a channel. The computation of a component (or of a sub-component) corresponds to the consumption of one single data token from each input channel and the production of one single data token on each output channel.

Since we want to use marked graphs also as a compact model to express the throughput and latency properties of a system (and its components), we make the following assumptions/restrictions with respect to the general definitions given in Section 2:

- Each marked graph $\mathcal{MG}$ is safe, live, consistent, and timed.
- Each transition in $\mathcal{MG}$ has unit delay, i.e.:

$$\forall t_i \in \mathcal{T}(\mathcal{MG}),\ d(t_i) = 1.$$

- For each marked graph, $\mathcal{MG}$ the firing rule (2) following Definition 2.1 is changed as follows: *an enabled transition always fires.*

Based on these assumptions, it is natural to think the behavior of the system represented by $\mathcal{MG}$ has an infinite sequence of atomic reactions: during each reaction all enable transitions do fire, thereby contributing to the enabling of transitions that will fire at the following reactions. Notice that the absence of a token at one of the incoming arcs of a transition (*vacancy*) is sufficient to keep the transition disabled (*AND firing rule*). It should not come as a surprise that this model presents several commonalities with the *synchronous programming model* [2], which has been proven quite effective as a simple way to access the power of concurrency in functional specification.

Given these assumptions, it is easy to see that as long as $\mathcal{MG}$ presents a cycle, the sequence of atomic reactions is periodic. The rate at which tokens occurs on a given arc (and correspondingly the rate at which a

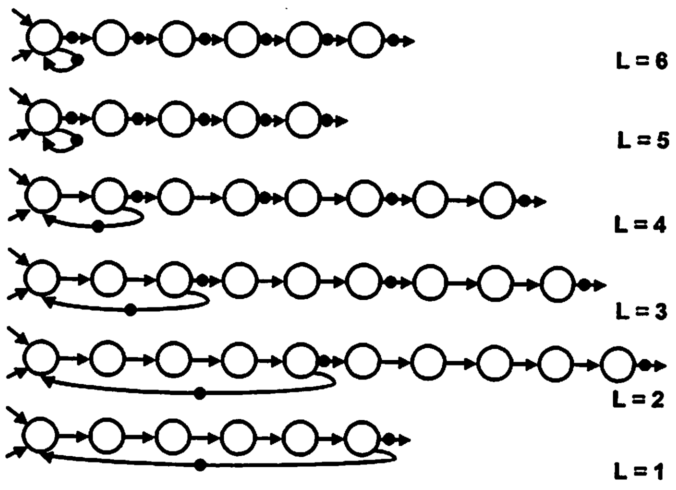

Figure 3: Instance MGs for Library Multipliers (target $\pi = 2ns$).

given transition fires) represents the throughput of the system modeled by $\mathcal{MG}$. This is denoted as $\vartheta(\mathcal{MG})$ and can be computed as the inverse of the minimum cycle time, i.e. $\vartheta(\mathcal{MG}) = \frac{1}{\tau_{min}(\mathcal{MG})}$. As a consequence, $\vartheta(\mathcal{MG})$ is a rational number in the interval $]0,1]$, In fact, the maximum achievable system throughput, equal to 1, is obtained when before and after each transition every place in the system contains one token or, equivalently, each firing in the firing sequence σ involves every transition in $\mathcal{MG}$. Since we assume that each transition has unit delay, Equation 1 computing the minimum cycle time of $\mathcal{MG}$ can be simplified as:

$$\tau_{min}(\mathcal{MG}) = \max_{c \in C(\mathcal{MG})} \left(\frac{|c|}{\mathcal{M}_0(c)} \right) \tag{2}$$

where $|c|$ denotes the number of vertices (transitions) on cycle c. For the marked graph $\mathcal{MG}$ of Figure 1 it is easy to verify by inspection that $\tau_{min}(\mathcal{MG}) = 2$ and, therefore, $\vartheta(\mathcal{MG}) = 0.5$. For a more complex graph $\mathcal{MG}$, one can build a companion graph G from $\mathcal{MG}$ as follows: (1) for each token of the marking $\mathcal{M}_0$ of $\mathcal{MG}$ build a vertex of G; (2) for each path connecting two tokens add an arc a_j into G to connect the corresponding vertices and set the weight $w(a_j)$ of the arc equal to the number of places traversed by this path. Then minimum cycle time $\tau_{min}(\mathcal{MG})$ coincides with the *maximum cycle mean* of G, which is defined as $mcm(G) = \max_{c \in G} \left(\frac{\sum_{a_j \in c} w(a_j)}{|c|} \right)$ and can be found using one of the many efficient algorithms that have been proposed, dating back to Karp's Algorithm [1, 5, 6].

3.1 Modeling IP Library Cores

SOC designers have access to various IP core libraries and, in general, each library offers more than one implementation for the same functionality module. For instance, designers who need to integrate one or more multipliers on their SOC may have to choose between a combinational multiplier or a pipelined multiplier, and, among the variety of pipelined multipliers, they may have to decide if they need a 3-stage pipelined multiplier of a 5-stage one. Sometimes IP cores are offered as so-called *hard* IPs, which are pre-synthesized blocks for a given technology. Often, designers use *soft* IP, that is RTL synthesizable specifications that will naturally lead to different results when synthesized using different technology libraries. We are interested in providing a

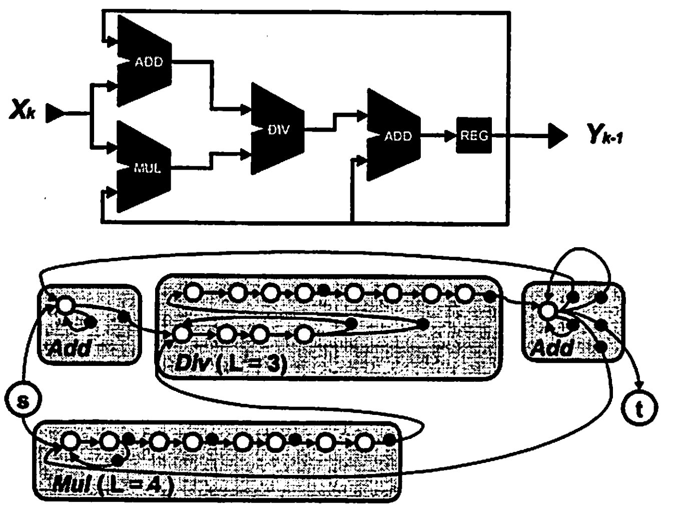

Figure 4: From Specification to Core Instantiation (target $\pi = 2ns$).

conceptual model for the characterization of the elements of an IP core library in terms of both the *maximum sustainable throughput* ϑ and the *minimum required latency* λ. The maximum sustainable throughput is the rate at which the IP core can be fed new data at its inputs. The minimum required latency is the number of clock cycles that pass by from the instant when the IP core samples new data at its input ports until the instant when it produces the corresponding results at its output ports. These figures may depend on several factors. In case of a library of soft IP cores, the logic synthesis step is likely among the most influential factors. Here, for the sake of simplicity, we assume to have a library of synthesized IP modules and we model them using only the two parameters that have the strongest impact on both ϑ and λ. They are:

- L: the *number of pipelined stages* of a sequential module. If the module is combinational then $L = 1$.
- δ: the *critical path* of the module. This is expressed in time units and represents a lower bound constraint on the period of the clock that can be used to run the module.

Now, for a given *target period* π of the design global clock, we can use the previous parameters to compute the maximum throughput ϑ, the minimum latency λ (expressed in number of global clock cycles), and the minimum absolute latency Λ (expressed in time units). This is done using the following expressions:

$$\vartheta = \left(\left\lceil \frac{\delta}{\pi} \right\rceil \right)^{-1} \tag{3}$$

$$\lambda = \frac{L}{\vartheta} \tag{4}$$

$$\Lambda = \lambda \cdot \pi \tag{5}$$

We want to clarify that, while writing the previous expressions, we assume that we are dealing with an ideal *nominal-design* situation. Naturally, more accurate expressions can be derived by, for instance, taking account of the various terms that contribute to the clock period overhead and subtracting from the target clock period the amount corresponding to clock skew, clock jitter, and latch overhead.

π	ϑ,λ	*Add*	*Multiplier*						*Divider*		
	L	1	1	2	3	4	5	6	1	2	3
	δ	1	11	9	6	4	2	1	12	10	7
1	ϑ	1	1/11	1/9	1/6	1/4	1/2	1	1/12	1/10	1/7
	λ	1	11	18	18	16	10	6	12	20	21
	Λ	1	11	18	18	16	10	6	12	20	21
2	ϑ	1	1/6	1/5	1/3	1/2	1	1	1/6	1/5	1/4
	λ	1	6	10	9	8	5	6	6	10	12
	Λ	2	12	20	18	16	10	12	12	20	24
3	ϑ	1	1/4	1/3	1/2	1/2	1	1	1/4	1/4	1/3
	λ	1	4	6	6	8	5	6	4	8	9
	Λ	3	12	18	18	24	15	18	12	24	27
4	ϑ	1	1/3	1/3	1/2	1	1	1	1/3	1/3	1/2
	λ	1	3	6	6	4	5	6	3	6	6
	Λ	4	12	24	24	16	20	24	12	24	24
8	ϑ	1	1/2	1/2	1	1	1	1	1/2	1/2	1
	λ	1	2	4	3	4	5	6	2	4	3
	Λ	8	16	32	24	32	40	48	16	32	24
12	ϑ	1	1	1	1	1	1	1	1	1	1
	λ	1	1	2	3	4	5	6	1	2	3
	Λ	12	12	24	36	48	60	72	12	24	36

Table 2: Throughput and latency values for the elements of an hypothetic IP library as function of target clock periods π.

Table 2 reports the latency and throughput values as functions of some possible target clock periods for the elements of an hypothetic IP core library. The library provides modules grouped in three distinct functional types: adders, multipliers, and divisors. We assume that, for an hypothetic technology process the library provides:

- one combinational adder with critical path $\delta \leq 1ns$;
- five distinct multipliers ranging from a combinational multiplier with critical path $\delta \leq 11ns$ to a 5-stage pipelined multiplier with critical path $\delta \leq 2ns$.
- three divisors: a combinational one with critical path $\delta \leq 12ns$, a 2-stage pipelined divisor with critical path $\delta \leq 10ns$, and a 3-stage pipelined divisor with critical path $\delta \leq 7ns$.

Each entry in Table 2 represents an example of IP core instance for a given target clock period π. For each core instance, we build an *instance marked graph* $\mathcal{IMG}$ that structurally captures the corresponding values of ϑ and λ by performing the following steps: (1) create a chain of vertices (transitions) that is as long as λ; (2) add a feedback arc from a vertex v in the chain back to the first vertex and distribute tokens on the arc of the resulting cycle c such that $\tau(c) = \vartheta^{-1}$; (3) distribute tokens on the arcs belonging to the path from v to the last vertex v' according to the value of ϑ. Figure 3 shows the $\mathcal{IMG}$s associated to the library multipliers for $\pi = 2ns$.

Now, let's assume that we need to use the IP cores of Table 2 to build a module that given a value x_0 and a sequence $a_1, \ldots, a_N$ computes a sequence of values x_k defined as:

$$x_k = x_0 + \sum_{j=1}^{k} \left(\frac{a_j + x_{j-1}}{a_j \cdot x_{j-1}} \right), \quad \forall k \in [1, N]$$

The block diagram at the top of Figure 4 illustrates a possible dataflow that implements the given specification using two adders, one multiplier, and one divisor. Then, let's assume that our first idea is to aim for a design with a target clock period of $2ns$. The bottom of Figure 4 illustrates the marked graph $\mathcal{MG}$ that we obtain by simply putting together the $\mathcal{IMG}s$ for the corresponding IP cores.

At this point, some considerations are in order:

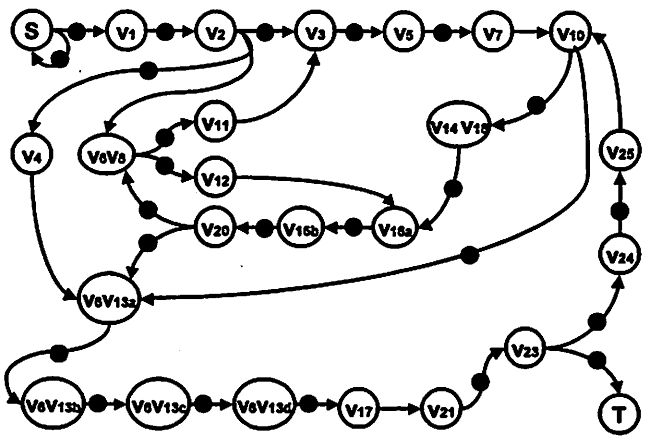

Figure 5: Equalized $\mathcal{MG}$ with $\vartheta(\mathcal{MG}) = 0.80$.

- First, it is obvious that in general the distributions of the tokens on $\mathcal{MG}$ may not be self-consistent because it is not necessary the case that the minimum cycle time of the resulting marked graph $\mathcal{MG}$ coincides with the minimum cycle times of each stand-alone component $I\mathcal{MG}$.
- As it is the case even for this simple example, the composition of $I\mathcal{MG}s$, generally, creates new cycles in additions to those already present in the stand-alone graphs. For instance, in Figure 4, we see that we have three additional cycles: one looping around the multiplier, the divisor, and the output adder; one looping around the two adders and the divisor, and, finally, a new self-loop around the output adder.
- Marked graph $\mathcal{MG}$ is a representation of the first-cut design targeting a clock period $\pi = 2ns$. By analyzing $\mathcal{MG}$ we can compute $\vartheta(\mathcal{MG}) = (\tau_{min}(\mathcal{MG}))^{-1} = 1/4$ and, therefore, realize that the effective clock frequency $\phi_{eff}(\mathcal{MG}) = \frac{\vartheta(\mathcal{MG})}{\pi} = 1/8ns = 125Mhz$. Also, we can compute the latency $\Lambda(\mathcal{MG}) = 2 \cdot 21 = 42ns$. What remains to be understood is whether this design is the best solution for the given technology and IP core library.

In the next sections, we will attempt to give answers to all these issues while discussing the throughput equalization problem.

4 Four Basic $\mathcal{MG}$ Transformations

From the previous section we learned how to use marked graphs to capture a complex system as a composition of simpler component and to express throughput and latency properties of both system and components. In this section we introduce a set of basic marked graph transformations that ultimately allow us to derive an alternative system implementation from a given one.

Let's consider a generic marked graph $\mathcal{MG}$. In general, $\mathcal{MG}$ will have multiple cycles and these cycles will intersect in various ways. We want to capture analytically how the vertices and the arcs are shared among intersecting cycles of $\mathcal{MG}$. Given a cycle $c \in C(\mathcal{MG})$, for all vertices $v_i \in \mathcal{MG}$ and all arcs $a_j \in \mathcal{MG}$, we define two binary variables:

- $x(v_i, c)$ is equal to 1 if $v_i \in c$, to 0 otherwise.
- $y(a_j, c)$ is equal to 1 if $a_j \in c$, to 0 otherwise.

Now, we can write the expression that gives the cycle time of a cycle $c \in \mathcal{MG}$ as follows:

Definition 4.1 *For a given initial marking $\mathcal{M}_0$, the* extended form *of the cycle time of cycle $c \in C(\mathcal{MG})$ is:*

$$\tau(c, \mathcal{M}_0) = \frac{\sum\limits_{v_i \in \mathcal{MG}} x(v_i, c) \cdot m(v_i)}{\sum\limits_{a_j \in \mathcal{MG}} y(a_j, c) \cdot n(a_j)} \tag{6}$$

where $\forall v_i, (m(v_i) = 1)$ and for all a_j, $n(a_j)$ is equal to one if a_j presents a token in the initial marking $\mathcal{M}_0$, otherwise it is equal to zero.

Expression 6 can be written for each cycle of $\mathcal{MG}$. For each vertex $v_i \in G$, the term $x(v_i, C) \cdot m(v_i)$ accounts for the latency contribution associated to v_i in case v_i belongs to c. Similarly, for each arc $a_i \in G$ the term $y(a_j, C) \cdot n(a_j)$ accounts for the throughput contribution associated to arc a_j if this arc belongs to c and if it presents a token.

From Equation 6, it is clear that if we remove a tokens from an arc a_j of c while leaving all the other arcs in $\mathcal{MG}$ unaffected, we end up increasing the cycle time of each cycle containing a_j. Furthermore, if a_j belongs to a critical cycle, we obviously increase the minimum cycle time of $\mathcal{MG}$, thereby affecting negatively its overall throughput. Conversely, if we find an arc a_j without a token and we put one on it, while leaving all the other arcs in $\mathcal{MG}$ unaffected, we decrease the cycle time of each cycle containing a_j. We call these two opposite transformations respectively *token addition* and *token removal*. We define now two other transformations that obtain similar results by operating on the numerator of Equation 6 instead of the denominator. While leaving the rest of $\mathcal{MG}$ we can singled out a vertex v_i and split it into two new ones. For all cycles c containing v_i, the effect is naturally to observe an increase of the value of $\tau(c, \mathcal{M}_0)$. We call this transformation *vertex splitting*. The reverse transformation, *vertex collapsing*, naturally consists in the removal of v_i and it is completed by connecting all the vertices directly feeding v_i with all the vertices that are directly fed by v_i. Differently from token addition and token removal, vertex splitting and vertex collapsing may change the graph structure. It is easy to see that the two vertices created from splitting v_i belong to the same cycles to which v_i used to belong. Differently from vertex splitting, vertex collapsing may also affect the graph cycle structure, because one or more cycles to which v_i used to belong can disappear as the connection between the vertices that were feeding v_i and those that were fed by v_i are merged.

These four basic $\mathcal{MG}$ transformations can be used independently or in combination. They clearly represent the simplest way of modeling the idea of pipelining/un-pipelining a component (or changing the number of stages of its pipeline) and adding/reducing communication latencies between components or sub-components.

We encode the $\mathcal{MG}$ transformation using a variable $m'(v_i)$ for each vertex $v_i \in \mathcal{MG}$ and a variable $n'(a_j)$ for each arc $a_j \in \mathcal{MG}$. These variables take values in the integer interval $[-1, 1]$ based on the following definitions:

- $m'(v_i) = -1$: collapse vertex v_i;
- $m'(v_i) = 0$: leave vertex v_i untouched;
- $m'(v_i) = 1$: split vertex v_i;
- $n'(a_j) = -1$: remove the token from arc a_j;
- $n'(a_j) = 0$: leave arc a_j untouched;

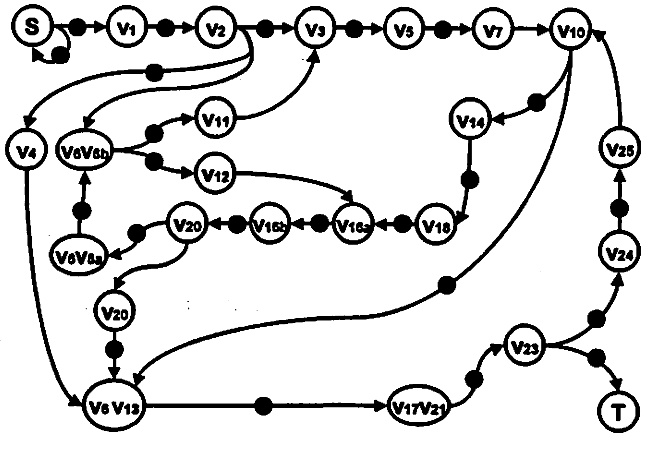

Figure 6: Equalized $\mathcal{MG}$ with $\vartheta(\mathcal{MG}) = 0.83$.

- $n'(a_j) = 1$: put the token on arc a_j;

Hence, the values of variables $m'(v_i), n'(a_j)$ unambiguously capture a specific combination of basic transformations on $\mathcal{MG}$, while the following expression givess exactly the resulting cycle time for each $c \in C(\mathcal{MG})$:

$$\tau(c, \mathcal{M}_0) = \frac{\sum_{v_i \in V} x(v_i, C) \cdot [m(v_i) + m'(v_i)]}{\sum_{a_j \in A} y(a_j, C) \cdot [n(a_j) + n'(a_j)]} .$$

Since the minimum cycle time is the inverse of the system throughput, this collection of four basic transformations represents a simple but effective toolkit to carry out both the performance analysis and design exploration of the system under the guidance of latency/throughput trade-offs. In, the following section we show how the definition of *throughput equalization problem* and its solution allows us perform these activities in an organized and effective manner.

5 The Throughput Equalization Problem

The throughput equalization problem is defined as follows.

Problem 5.1 (Throughput Equalization Problem)

Given: An $\mathcal{MG}$ with an initial marking $\mathcal{M}_0$, $|V|$ vertices, $|A|$ arcs, and K cycles.

Minimize: The cost $C = \sum_{v_i \in V} |m'(v_i)| + \sum_{a_j \in A} |n'(a_j)|$ over all integer variables $m'(v_1), \dots, m'(v_{|V|})$ and $n'(a_1), \dots, n'(a_{|A|})$, with $\forall i \in [1, |V|], (m'(v_i) \in [-1, 1]$ and $\forall j \in [1, |A|], (n'(a_j) \in [-1, 1]$.

Subject to: $\exists q \in \mathbf{Q}^+, q \geq 1,\ \forall k \in [1,K]$:

$$\frac{\sum_{v_i \in V} x(v_i, C_k) \cdot [m(v_i) + m'(v_i)]}{\sum_{a_j \in A} y(a_j, C_k) \cdot [n(a_j) + n'(a_j)]} = q$$

It is easy to see that Problem 5.1 has always a solution. In fact, the trivial variable configuration that is obtained by setting $\forall v_i,\ (m'(v_i) = 0)$ and $\forall a_j,\ (n'(a_j) = 1 \Leftrightarrow n(a_j) = 0)$ always satisfies the problem constraints with $q = 1$. Notice also that each configuration of the integer variables $m'(v_1), \ldots, m'(v_{|V|})$ and $n'(a_1), \ldots, n'(a_{|A|})$ unambiguously leads us to derive a new graph $\mathcal{MG}$ whose cycles have all cycle times equal to the corresponding value of q. The cost function that we use in the above definitions is a very simple one and it is motivated by finding the throughput equalization while applying the minimum amount of $\mathcal{MG}$ transformations from the given design. Hence, this cost function is ideal when the problem is solved as an intermediate step for an interactive design exploration engine. More sophisticated cost functions can obviously be defined. In any case, algorithm *RateEqualizer* illustrated in Figure 8 not only solves Problem 5.1 exactly, but may also be used to return all the variable configurations that satisfy some input problem constraints specified in terms of the desired throughput range and the maximum number of transformations allowed.

The algorithm is organized on three basic routines. The main routine *RateEqualizer* invokes routine *RelaxedRateEqualizer* to find all the candidate solutions of a relaxed version of Problem 5.1. The relaxation consists in the fact that routine *RelaxedRateEqualizer* assumes that all the cycles have pairwise empty arc/vertex intersection. In other words, routine *RelaxedRateEqualizer* would give the exact solution if the marked graph was a connection of distinct strongly connected components (SCCs) each having at most one cycle. Notice that *RelaxedRateEqualizer* searches the solution space in a very efficient way because the independence among the variables $(m'_1, n'_1), (m'_2, n'_2), \ldots, (m'_K, n'_K)$ does not make necessary to perform any branch-and-bound technique. Once all the candidate solutions are returned to *RateEqualizer* the core routine *RecursiveStep* is invoked for each candidate solution. Routine *RecursiveStep* attempts to *justify* the candidate solution by finding an assignment $m'(v_1), \ldots, m'(v_{|V|})$ and $n'(a_1), \ldots, n'(a_{|A|})$, that satisfies the ratio found during the solution of the relaxed problem without violating the reciprocal constraints that cycles sharing common variables impose on each other. The search is limited by the maximum number of transformations allowed. All the valid solutions are returned in increasing order of minimum cycle time.

Example 5.1 Figure 7 reports two alternative implementations for the design of Figure 4 that have been obtained automatically by running the *RateEqualizer* algorithm. These are equalized implementations in the sense that all cycles in the corresponding $\mathcal{MG}$s have the same cycle throughput $\vartheta(\mathcal{MG})$: this is equal to $1/3$ for the top design and $2/3$ for the bottom design.

Let's analyze first the top design: by studying the charactersistics of the $\mathcal{IMG}$ for the instances of the IP library cores, we can see that it is suggested the use of a 3-stage pipelined divisor with latency between 9 and 13 clock cycles and a 3-stage pipelined multiplier with latency between 6 and 8 clock cycles. Using equations 3 and observing the IP library illustrated in Table 2, we deduce that the optimum feasible clock period is between 2.5ns and 3ns. This is worse than the target clock period of our first-cut design that was 2ns. However, besides having a well-balanced design, it turns out that by running the system with a clock of 2.5ns we have an effective clock frequency of $\phi'_{eff}(\mathcal{MG}) = \frac{\vartheta(\mathcal{MG})}{\pi} = \frac{1}{3} \cdot \frac{1}{2.5ns} = \frac{1}{7.5ns} = 133Mhz$ which is better than the $125Mhz$ figure of our unbalanced first-cut design.

Now, let's analyze the bottom design: first we notice that we need to pipeline the adders by 1 stage. This can be done as long as we use a *clock* period longer than $2ns$. Then, it is suggested the use of a 2-stage pipelined divisor with latency between 1 and 5 clock cycles and a 4-stage pipelined multiplier with latency between 3 and 5 clock cycles. This constrains us to have a clock period longer than $5ns$. However, our balanced design presents again an effective clock frequency of $\phi''_{eff}(\mathcal{MG}) = \frac{\vartheta(\mathcal{MG})}{\pi} = \frac{2}{3} \cdot \frac{1}{5ns} = \frac{1}{7.5ns} = 133Mhz$ which is again better than

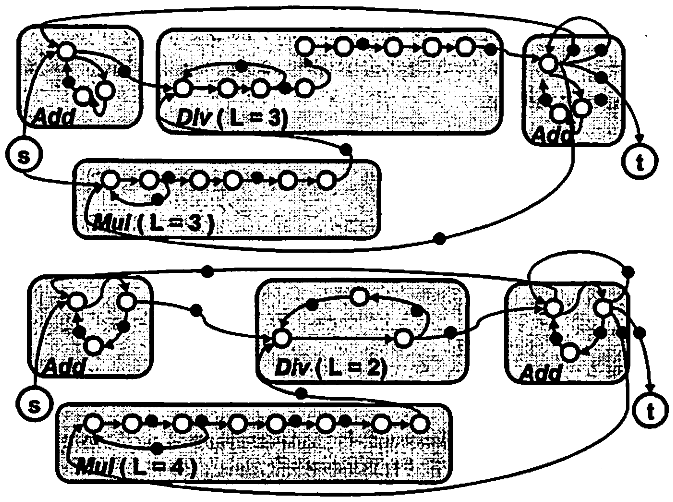

Figure 7: Two Equalized Implementation for design of Figure 4. The top design has $\vartheta(\mathcal{MG}) = \frac{1}{3}$ with $\pi = 2.5ns$. The bottom design has $\vartheta(\mathcal{MG}) = \frac{2}{3}$ with $\pi = 8ns$.

$125Mhz$. Finally, as we compare the latency of the two systems, we see that the top design has a better absolute latency $\Lambda(\mathcal{MG}) = 7 \cdot 2.5ns = 17.5ns$ with respect to both the first-cut design (having $\Lambda(\mathcal{MG}) = 42ns$) and the bottom design (having $\Lambda(\mathcal{MG}) = 9 \cdot 5ns = 45ns$).

Example 5.2 Let's apply the *RateEqualizer* algorithm also to the $\mathcal{MG}$ of Figure 1 having $\vartheta(\mathcal{MG}) = 0.5$. In this graphs some vertices represent latency cycles due to the inter-communication between system components. Here we need to equalize the cycle time of the six cycles of Table 1. The trivial solution is obtained by equalizing all these cycles to the value 1 by inserting tokens on every arc. The *RateEqualizer* algorithm finds two more interesting solutions. The solution of Figure 5 has cost equal to 9 for a system throughput $\vartheta(\mathcal{MG}') = 0.8$. This is obtained by pipelining vertex v_{16}, pipelining three times vertex v_{13}, collapsing vertices v_{14} (into v_{18}) and v_6 (into both v_8 and v_{13a}) and, finally, reducing the latency of the connection from v_{18} to v_{13} by three. Instead, the solution of Figure 6 has the optimum cost (equal to 7) for a system throughput $\vartheta(\mathcal{MG}'') = 0.83$. This is obtained by pipelining vertex v_{16} and vertex v_8 while collapsing v_6, and reducing the latency of the connection from v_{18} to v_{13} by three and the one from vertex v_{13} to v_{21} by one. Notice that both these solutions reduce the total number of cycles in the graph from 6 to 4. This may be an indication that the original design didn't have a good balance between communication and computation.

6 Conclusions

We presented a novel formal model that acts as a *common semantic domain* to express on one side the latency/throughput properties of IP library components, and, on the other side, the latency/throughput trade-offs of the various system implementations that can be obtained by composing modules that are instanced from the IP library. Other important contributions are the definition of the system throughput equalization problem and the derivation of the *RateEqualizer* algorithm that solves it exactly. The natural next step in this research is to develop an interactive design environment for system-on-chip. SOC designers are currently facing a difficult *performance balancing problem* when they must choose and assemble several IP modules from different vendor libraries. The goal of the new design environment will be to facilitate the exploration of latency/throughput trade-offs at early stages of the design cycle. We trust that the techniques proposed in this paper will represent a key part of the system, because it provides a theoretically-sound method to tackle the problem globally, with means other than simulation and intuition.

Acknowledgments

Discussions with Luciano Lavagno and Fernando De Bernardinis are gratefully acknowledged.

References

[1] F. Baccelli, G. Cohen, G.J. Olsder, and J.P. Quadrat. *Synchronization and Linearity*. Wiley, New York, 1992.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Language Twelve Years Later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[3] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A Methodology for "Correct-by-Construction" Latency Insensitive Design. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 309–315. IEEE, November 1999.

[4] J.A. Darringer, R. A. Bergamaschi, S. Bhattacharya, D. Brand, A. Herkersdorf, J. K. Morrell, I. I. Nair, and P. Sagmeister. Early Analysis Tools for System-on-a-Chip Design. *IBM J. Res. Develop.*, 46(6):691–707, November 2002.

[5] A. Dasdan and R. K. Gupta. Faster Maximum and Minimum Mean Cycle Algorithms for System-Performance Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, October 1998.

[6] R. M. Karp. A Characterization of the Minimum Cycle Mean in a Digraph. *Discrete Math.*, 23:309–311, 1978.

[7] K.S. Khouri, G. Lakkshminarayana, and N.K. Jha. High-level synthesis of low-power control-flow intensive circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(12):1715–1729, December 1999.

[8] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.

[9] T. Lin and L. T. Pileggi. Throughput-driven ic communication fabric synthesis. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 274–279, 2002.

[10] A. Mathur, A. Dasdan, and R. K. Gupta. Rate Analysis for Embedded Systems. *ACM Trans. on Design Automation of Electronic Systems*, 3(3):408–436, July 1998.

[11] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[12] H. Shah, P. Shiu, B. Bell, M. Aldredge, N. Sopory, and J. Davis. Repeater insertion and wire sizing optimization for throughput-centric vlsi global interconnect. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 280–284, 2002.

[13] N. Shenoy. Retiming: Theory and Practice. *Integration, the VLSI Journal*, 22:1–21, 1997.

```
 1: $RateEqualizer\left(m(v_1),\ldots,m(v_{|V|}),n(a_1),\ldots,n(a_{|A|})\right)$,
 2: for all $k \in [1,K]$ do
 3:   $m_k \Leftarrow \sum_{v_i \in V} x(v_i,C_k)\cdot m(v_i)$;  $n_k \Leftarrow \sum_{a_j \in A} y(a_j,C_k)\cdot n(a_j)$
 4: end for
 5: $\mathcal{CES}_{sort} \Leftarrow RelaxedRateEqualizer\ (\ (m_1,n_1),\ldots,(m_K,n_K)\ )$
 6: for all candidate $c \in \mathcal{CES}_{sort}$ do
 7:   {Reset set of auxiliary variables}
 8:   for all $v_i \in V, a_j \in A$ do
 9:     $m'(v_i) \Leftarrow 0$; $b(v_i) \Leftarrow false$; $n'(a_j) \Leftarrow 0$; $b(a_j) \Leftarrow false$;
10:   end for
11:   $(M,N) \Leftarrow computeTargetValues(c)$
12:   {Sort cycle row indexes in increasing order of their targets}
13:   $\mathcal{H} \Leftarrow sortCycleRowIndexes(M,N)$
14:   {Fire recursive step to seek new equalization solution}
15:   if $(recursiveStep(\mathcal{H},0) = true)$ then
16:     $s \Leftarrow SaveSolution\ (m(v_1),\ldots,m(v_{|V|}),n(a_1),\ldots,n(a_{|A|}))$
17:     $\mathcal{S} \Leftarrow \mathcal{S} \cup s$
18:   end if
19: end for
 1: $RelaxedRateEqualizer\ (\ (m_1,n_1),\ldots,(m_K,n_K)\ )$
 2: {Find $(m'_1 n'_1),\ldots,(m'_K,n'_K) \in [-1,1] \times [-1,1]$ s.t.}
 3: {$\exists q \in \mathbb{Q}^+,\ \forall k \in [1,K],\ (\ \frac{m_k+m'_k}{n_k+n'_k} = q\ )$...}
 4: {...and the cost $C = \sum_{k=1}^{K} |m'_k| + |n'_k|$ is minimum.}
 5: $C_{ub} \Leftarrow 0$
 6: for $(k = 1; k \leq K; k{+}{+})$ do
 7:   $C_{ub} \Leftarrow C_{ub} + |n_k|$
 8: end for
 9: {Sort $p_k = (m_k,n_k)$ in ascending order by ratio $\frac{m_k}{m_k}$}
10: $\mathcal{P} \Leftarrow sortInputPairsByRatio(\ (m_1,n_1),\ldots,(m_K,n_K)\ )$
11: {Find the candidate equalization seeds}
12: $\mathcal{CES} \Leftarrow findCandidateEqualizationSeeds(\mathcal{P},C_{ub})$
13: {Sort the candidate equalization seeds by their optimality}
14: return $\mathcal{CES}_{sort} \Leftarrow sortCandidateEqualizationSeeds(\mathcal{CES})$
 1: $RecursiveStep\ (\mathcal{H},h)$
 2: if $(h = K)$ then
 3:   {No more rows to equalize. Exit recursion successfully.}
 4:   return $true$
 5: end if
 6: $k \Leftarrow \mathcal{H}[h]$
 7: $MM[k] \Leftarrow M[k] - \sum_{v_i \in V} x(v_i,C_k)\cdot m'(v_i)$
 8: $NN[k] \Leftarrow N[k] - \sum_{v_i \in V} x(v_i,C_k)\cdot m'(v_i)$
 9: $\mathcal{R} \Leftarrow findSetOfConfigurations(MM[k],NN[k])$
10: for all $r \in \mathcal{R}$ do
11:   {Pick configuration r as a possible cycle solution}
12:   for all $v_i,a_j \in C_k$ do
13:     $(m'_s(v_i),n'_s(a_j)) \Leftarrow (m'(v_i),n'(a_j))$
14:     $(m'(v_i),n'(a_j)) \Leftarrow setVariablesFromConfiguration(r)$
15:     $b_s(v_i) \Leftarrow b(v_i)$; $b_s(a_j) \Leftarrow b(a_j)$;
16:   end for
17:   if $(recursiveStep(\mathcal{H},h{+}{+}) = true)$ then
18:     return $true$
19:   else
20:     for all $v_i,a_j \in C_k$ do
21:       $(m'(v_i),n'(a_j)) \Leftarrow (m'_s(v_i),n'_s(a_j))$
22:       $b(v_i) \Leftarrow b_s(v_i)$; $b(a_j) \Leftarrow b_s(a_j)$;
23:     end for
24:   end if
25: end for
26: return $false$
```

Figure 8: Algorithm to solve the Rate Equalization Problem