Topology-Based Performance Analysis and Optimization of Latency-Insensitive Systems

Rebecca L. Collins and Luca P. Carloni, *Member, IEEE*

Abstract

Latency-insensitive protocols allow system-on-chip engineers to decouple the design of the computing cores from the design of the inter-core communication channels while following the synchronous design paradigm. In a latency-insensitive system (LIS) each core is encapsulated within a shell, a synthesized interface module that dynamically controls its operation. At each clock period, if new data has not arrived on an input channel or a stalling request has arrived on an output channel, the shell stalls the core and buffers other incoming valid data for future processing. The combination of finite buffers and backpressure from stalling can cause throughput degradation. Previous works addressed this problem by increasing buffer space to reduce the backpressure requests or inserting extra buffering to balance the channel latency around a LIS. We explore the theoretical complexity of these approaches and propose a heuristic algorithm for efficient queue sizing. We also practically characterize several LIS topologies and how the topology of a LIS can impact not only how much throughput degradation will occur, but also the difficulty of finding optimal queue sizing solutions.

Index Terms

Latency-Insensitive Design, Performance Analysis, Systems-on-Chip. System-Level Design

I. Introduction

LATENCY-INSENSITIVE design (LID) [6], [7] is a correct-by-construction methodology for systems-on-chip (SOCs) that simplifies the assembly of intellectual property (IP) cores by reconciling the traditional methods for digital chips based on the *synchronous paradigm* [4] with the dominant impact of interconnect delay that characterize nanometer technologies [9]. In particular, LID decouples the design of the IP cores from the design of the communication channels among them. Also, for the latter it eases the application of *wire pipelining*, a technique to fix timing violations in global interconnect that is both effective and challenging [2], [36].

Given a netlist of IP cores, which may be specified as synthesizable RTL modules, a latency-insensitive system (LIS) is automatically derived by encapsulating each core within a *shell*. A shell is a synthesized logic block that implements a latency-insensitive protocol and acts as an interface around the core for global, inter-core, communication. The idea is to build a distributed global communication infrastructure that relies on a set of point-to-point, lossless, elastic, pipelined channels instead of centralized communication resources. IP cores may be synchronous sequential logic blocks of any complexity as long as they satisfy the *stallability* requirement; i.e., their operation can be temporarily stalled through *clock-gating*. Inter-shell channels made of long wires can be pipelined through the insertion of *relay stations* (clocked buffers with two-fold storage capacity [6]) in order to meet the target clock period. The theory of LID guarantees that *any* number of relay stations can be distributed on these channels up to late stages of the design process without requiring the re-design of any IP core and without jeopardizing the system behavior [7]. Essentially, this is possible because: (a) the data exchanged by the shells are marked as either valid [1] or void, (b) the relay stations are initialized with void data, and (c) each shell keeps its core unaware of the existence of void data by controlling it via an *AND-firing* policy: at each clock period the

R. L. Collins and L. P. Carloni are with the Department of Computer Science, Columbia University in the City of New York, New York, NY 10027, ({rlc2119,luca}@cs.columbia.edu).

[1]Valid and void data are also denoted respectively as informative and stalling events in the theory of latency-insensitive design [7].

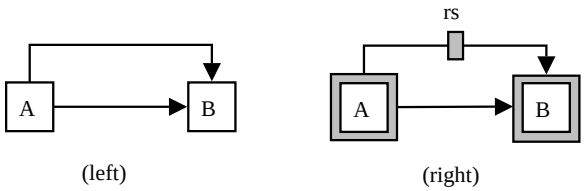

Fig. 1. Example of a system transformed into a LIS. A and B are encapsulated in shells, and a relay station is inserted on A's upper channel.

component	t_0	t_1	t_2	t_3	...
A (upper)	0	2	4	6	...
A (lower)	1	3	5	7	...
B	0	τ	1	5	...
Relay Station	τ	0	2	4	...

TABLE I
OUTPUT TRACES OF THE COMPONENTS IN THE LIS OF FIG. 1(RIGHT).

shell fires the core if and only if it has a new valid data from each input channel, and it stalls the core otherwise. Valid data that are not consumed while the core is stalled are buffered by input *queues* (a shell has a distinct input queue per each channel). As a result, the behavior of the LIS is *latency-equivalent* to the behavior of the original synchronous system; i.e., each channel presents exactly the same sequence of valid data but for the possible interleaving of some void data [7].

The simple example in Figure 1 illustrates how a synchronous system is transformed into a LIS. Each of the two IP cores A and B is encapsulated in a shell. Let's assume that the upper channel has been routed on a path much longer than the lower channel and, therefore, in order to meet the target clock period we must pipeline it by inserting one relay station rs. Then, Table I illustrates a behavior of this simple system where A is a module that generates even numbers to its upper channel and odd numbers to its lower channel and B is an adder whose latched output is initialized to zero. We use τ to denote a void data item as proposed in [7].

Besides the "traditional" clock frequency of its synchronous circuits, the main performance metric of a LIS is the rate at which it processes valid data [8]. This throughput, which may be reduced by the periodic occurrence of void data, depends on two factors: (1) the internal structure of the LIS and (2) the interaction with the environment where the LIS operates. The internal structure determines its *maximal sustainable throughput (MST)* θ as the LIS effectively processes valid data at this rate unless the environment forces it to slow down (e.g., by not providing enough valid data). The insertion of a relay station on a *feedback loop* of a LIS reduces its MST because the initialization value τ continues to circulate around the loop and causes each shell on the loop to periodically stall its core [8], [27]. As explained in Section III, LISs can be effectively modeled with *marked graphs* and, in particular, the MST of a LIS can be precisely derived by performing a static analysis of the structure of the corresponding marked graph.

In the example of Figure 1 there is no feedback loop and the τ value that is initially present in the relay station eventually leaves the system, which therefore has the highest possible MST, i.e. $\theta = \frac{\#\ valid\ data\ items}{clock\ periods} = 1$. Note, however, that the presence of the void data item forces the shell of B to stall its core during the first clock period. Hence, this shell must buffer the first valid data on A's lower channel (equal to 1) in the corresponding input queue while waiting for the first valid data on A's upper channel (equal to 0) to traverse the relay station. If this simple system does not interact with the environment, a queue of size one provides sufficient storage space to avoid any data loss.

In general, however, systems are combined to derive more complex systems: this makes it impossible to know in advance the sequence of τ data items that each component will observe during its operations. For instance, if an uplink subsystem with an MST of $\frac{3}{4}$ feeds another downlink subsystem with a lower MST

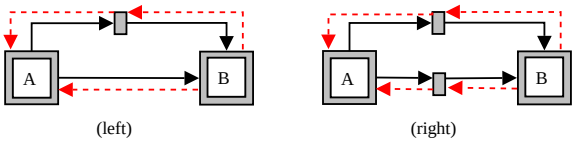

Fig. 2. Adding backedges to the LIS example (left). Inserting an additional relay station for performance reasons (right).

of $\frac{2}{3}$, only the presence of queues of infinite size (*infinite queues*) could provide the shells of the latter with sufficient buffering capacity. But since infinite queues are unrealizable in practice, a communication protocol is necessary among the shells to avoid any possible loss of valid data. Specifically, a downlink shell must be able to send a *stop* signal back on an input channel to indicate that its queue is full and that the corresponding uplink shell must stall. This operation, called *backpressure* [7], guarantees lossless communication. But its implementation, which is based on the addition of a backward communication line on each channel, may cause the introduction of new feedback loops across multiple shells that in turn may force the overall LIS to have a degraded MST.

In Figure 2(left) we illustrate backpressure by adding a backward edge (*backedge*) for every forward edge in our example. This causes the introduction of two backpressure feedback loops. Each of these loops comprises some forward and some backward edges. Now, if we suppose that the shells have queues with fixed capacity $q = 1$, this system's MST becomes $\frac{2}{3}$. Note that even though the shell of B has space to store one data token from A, it still must send a stop signal to A on the lower channel as it fills the space because it does not know beforehand when valid data will arrive. In other words, if B receives a τ on the upper channel and a new valid data token on the lower channel when the lower input channel queue is already full, then the valid data token would be lost.

Marked graphs can be used to model both *ideal* (i.e. theoretical and not realizable) LISs with infinite queues and *practical* LISs that use finite queues together with backpressure. If G denotes a marked graph modeling an ideal LIS, $\theta(G)$ denotes G's MST, and $d[G]$ is the marked graph obtained by adding backedges to G (the *doubled graph* of G). It has been shown that $\theta\big(d[G]\big) = \theta(G)$ when the system has finite queues that are "big enough" [27]. Still, it is a challenge to determine how big the finite queues must be to match the performance of a system with infinite queues (*queue sizing problem*).

In some cases an alternative to increasing queue size is to insert *additional relay stations* that would not be required for wire pipelining purposes but that are useful to increase the value of $\theta(d[G])$, possibly up to $\theta(G)$. In fact, for the example of Figure 2(right) it is sufficient to insert a relay station on the lower channel so that A's data are delayed one period along both channels, and B receives data from both of them at the same time. With respect to increasing the queue sizes, this technique allows more flexible placement of the additional storage space. However, as we show in Section VI, it does not work for all possible cases because the additional relay stations can potentially impact performance elsewhere in the system.

Contributions. We focus on the performance optimization of the *practical* LIS (with backpressure and finite queues) so that its MST is equal to the *ideal* MST of the equivalent *theoretical* LIS (with infinite queues and no backpressure). In other words, we study the problem of how to avoid *throughput degradation* in LIS implementations that are based on backpressure. We consider both the optimal sizing of the input queues in the shells and the insertion of additional relay stations beyond what is required for wire pipelining purposes. We provide a unifying modeling framework for this problem based on marked graphs (Section III) and we outline which approaches work for different classes of LIS topologies. In some cases, fixed queue sizing is enough to optimally solve MST degradation from backpressure (Section IV). In the most general case, however, no easy solution exists for optimally sizing the queues. In fact, we prove that this is an NP-complete problem (Section V). On the other hand, as we contrast queue sizing with the alternative method of relay-station insertion, we demonstrate that the latter has more limited applicability by presenting the counter-example of a LIS whose MST cannot be optimized by only adding

relay stations (Section VI). Further, we prove that optimal relay station insertion is also NP-complete (Appendix). Finally, we propose a heuristic algorithm for the queue sizing problem (Section VII) and evaluate empirically how well it performs compared to an exact algorithm (Section VIII).

II. Related Work

Wire pipelining, i.e. the insertion of sequential elements (or clocked buffers) to pipeline long wires in integrated circuits that are designed with nanometer technologies, has been discussed in several works [6], [15], [23], [28], [32], [36]. Variations of a relay station circuit have been used for wire pipelining to build efficient on-chip global communication infrastructures in various projects [2], [3], [13], [14], [20].

The performance analysis of latency-insensitive systems originally presented in [8] is based on the assumption of infinite queues. With infinite queues, backpressure mechanisms are not necessary, and the MST of a LIS is always at its ideal limit. More recent works recognize the necessity of backpressure in practical LIS implementations and explore ways to deal with the throughput degradation that can occur. In particular, Lu and Koh show that the performance of a practical LIS with finite queues can match the performance of an ideal LIS with infinite queues if the queues are big enough [26], [27]. In order to find optimal queue sizes, they employ mixed integer linear programming.

Casu and Macchiarulo avoid queue sizing issues by scheduling the core firing and eliminating back-pressure [11], [12]. This technique works when it is possible to analyze statically how the behavior of the global system should be scheduled throughout its components, but it cannot be applied to open systems that operate in an environment that may produce data at a dynamically variable rate. Casu and Macchiarulo [10] are also the first to propose solving throughput degradation by inserting additional relay stations to balance the latencies of converging communication paths (like the two paths in the example of Figure 2). In Section VI we discuss this technique and contribute the example of a LIS where this approach alone cannot bring about a full recovery of the ideal throughput.

In order to study how to avoid MST degradation in a LIS we formally define the Queue Sizing Problem and Relay Station Insertion Problem. These problems are somewhat related to the Slack Matching Problem that has been defined for quasi delay-insensitive (QDI) *asynchronous* systems [29], [33]. With slack matching, paths in an asynchronous system are pipelined to meet a target throughput goal. In a LIS, which is a synchronous system, this technique is akin to breaking up a core-shell pair into multiple core-shell pairs. With queue sizing, however, we do not break up core/shell pairs, but we simply add extra storage capacity on the backpressure paths. And with relay station insertion, we pipeline wires between computational cores, but not the core logic itself. The Slack Matching Problem has been modeled with marked graphs and proven NP-complete by Kim and Beerel [25] and has been solved with algorithms that are based on mixed integer linear programming by Prakash and Martin [33].

In this work, we forgo the popular mixed integer linear programming approach to these hard problems and instead we analyze the system topology to identify special cases where the problem is not as difficult. In addition, we extend our previous results [16] on throughput degradation in LISs with proofs about the complexity of optimal queue sizing and relay station insertion.

III. Modeling a LIS with Marked Graphs

Marked graphs, also known as *decision-free* Petri nets, are a simple model for concurrent systems [18] and, particularly, for systems that have a periodic behavior. Their simplicity makes them quite amenable to analysis.

The components of a LIS produce valid/void data synchronously according to a global clock. LISs can be conveniently modeled with marked graphs at the communication-protocol level because (a) they operate as deterministic systems and (b) it is only necessary to distinguish valid from void data regardless of the specific value of the valid data items.

A. Marked Graphs

Formally, a *marked graph* is a tuple $G = (P, T, F, M_0)$ where P is a finite set of *places*, T is a finite set of *transitions*, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, $M_0 : P \to \mathbf{Z}^*$ is the initial marking (or state), and such that $P \cap T = \emptyset \ \wedge \ P \cup T \neq \emptyset$ and $\forall p \in P\big(|\{t|(t,p) \in F\}| = |\{t|(p,t) \in F\}| = 1\big)$.

In other words, a marked graph is a bipartite directed graph with two kinds of vertices (places and transitions) where each place has exactly one incoming edge and one outgoing edge that both go to transitions. Places can hold zero or more *tokens*; transitions cannot hold tokens, but they can *fire*. A firing creates a new marking by moving tokens around in the graph. A transition is *enabled* to fire when the place on each of its incoming edges has at least one token. When a transition fires, it takes a token from each of its incoming places and puts a new token into each of its outgoing places [18]. The *initial marking* of a marked graph specifies how many tokens each place has before any firing. We report here some of the many important properties of marked graphs. For proofs and more complete discussions, the reader is invited to consult the extensive literature on the subject [1], [18], [30], [31].

While the firing activity may change the overall number of tokens in a marked graph G, the number $M_0(c)$ of tokens that are present on a cycle c of G is invariant under any firing sequence. If G is strongly-connected, then a firing sequence leads G back to the initial marking M_0 when it fires every transition an equal number of times.

A marked graph G is *timed* if there exists a delay $d(t)$ associated with each transition. The *cycle mean*, or *cycle metric*, $\mu(c)$ of a cycle c of G is the sum of the transition delays along a cycle divided by the number of tokens in the cycle, i.e:

$$\mu(c) = \frac{\sum_{t \in c} d(t)}{M_0(c)}$$

The *cycle time* $\pi(t)$ of a transition t of G is the average time separation between two consecutive firings of t. Its reciprocal gives the average firing rate of t.

If G is strongly connected then all transitions have the same cycle time $\pi(G)$, which is called the cycle time of G and is equal to the largest cycle mean across all its cycles [34]. [2] Cycles whose cycle mean coincide with $\pi(G)$ are called *critical cycles*. Cycle time $\pi(G)$ is a natural performance metric for the system modeled by G because its reciprocal gives the rate of consumption/production of tokens, i.e. the system's throughput. It can be computed using Karp's algorithm to find the maximum cycle mean of a directed graph [21], [24] or using linear programming [5], [37].

B. Modeling Latency-Insensitive Systems with Marked Graphs

Since LIS are synchronous systems, we model them using timed marked graphs such that $\forall t \in G\ \big(d(t) = 1\big)$. [3] Also, for our purposes we slightly restrict the behavior of a marked graph by assuming that it occurs as an indexed sequence of markings according to a *step semantics*: the marked graph moves from a marking M_i to a marking M_{i+1} in a single step during which all enabled transitions fire concurrently. Given this assumption, the firing activity of a timed marked graph can be cast into the synchronous paradigm as it is discussed in [4]: it evolves through an infinite sequence of atomic reactions where each reaction corresponds to a step between two markings and can be indexed with a natural number capturing the progression of time (a *timestamp* or clock period).

Figure 3 shows how we use marked graphs to model a relay station and a two-input shell with backpressure. The large white circles represent places, the small black dots (in the white circles) represent tokens, and the thin black rectangles represent transitions. Each token on a forward edges models a valid data on a LIS channel. Conversely, each token on a backedge (shown as a dashed line) represents one available slot in a queue or a relay station. In the initial marking, the relay station's incoming forward

[2] Similar results are found in [5], [21], [30], [35].

[3] Given this assumption of unit transition delay, the numerator of the cycle mean coincides with the number of transitions in the cycle, which is also equal to the number of places. Hence the cycle mean becomes equal to the ratio of places and tokens around the cycle.

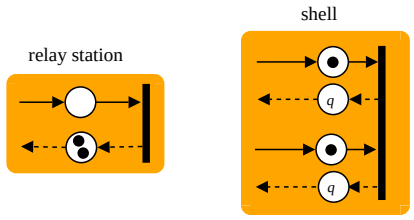

Fig. 3. Marked-graph models of relay stations and shells with backpressure.

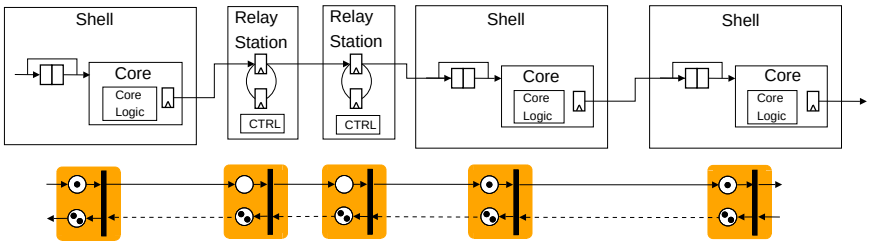

Fig. 4. Marked-graph model (with $q = 2$) of a path across multiple shells and relay stations in a LIS.

edge has no token since it must produce a τ in the first timestamp and its outgoing backedge has two tokens corresponding to the two available slots in the queue. The shell's incoming forward edges have one token each since a shell produces a valid data token in the first timestamp, and its backedges have a number q of tokens that is equal to the capacity of the corresponding input queue.

Figure 4 shows a path across multiple shells and relay stations in an RTL implementation of a LIS and the corresponding path in a marked-graph model with $q = 2$. To avoid cluttering in the RTL diagram we do not show the backpressure signals and we only show the single relevant input channel in the shells. Recall that compared with a simple edge-triggered flip-flop, which can be used to pipeline channels but without backpressure, a relay station presents the characteristic twofold buffering capability (together with the necessary control logic), thereby a *secondary* (or *auxiliary*) register is coupled to a *main* register [6] Also, a shell relies on the logic of its stallable core to latch the output signals and features by-passable input queues to avoid adding any delay to the original latency of a core when stalling is not necessary. In the best case, i.e. in the absence of any stalling, the latency to traverse either a relay station or a shell-core pair is one clock period. [4] In the marked-graph model the various data storage elements in each module are abstracted to a single place per shell or relay station that can hold multiple tokens when stalling occurs. When the marked graph is initialized, we place the data tokens that will be transferred during the first clock period behind the transition corresponding to the shell that is initialized with this data.

Due to the structure of relay stations and shells, the structure of a marked graph modeling a LIS is a little more restricted than that of a general marked graph. Specifically, with respect to the initial marking: (a) if a transition has an incoming place with one token, then that transition corresponds to a shell in the LIS and all of its incoming places must have one token; and (b) if a transition has an incoming place with zero tokens, that transition corresponds to a relay station in the LIS and it must have only one incoming and one outgoing place. Also notice that places on forward edges have either one or zero tokens and that every cycle must have at least one token.

C. Maximal Sustainable Throughput

The fundamental performance metric of a LIS is the rate of production of valid data, i.e. its throughput. The throughput of a LIS depends on two factors: its internal structure and its interaction with the

[4]More precisely, in the absence of stalling the latency to traverse a shell-core pair is the same as the latency to traverse the sole core, which may be greater than one if the core is a pipelined circuit like a three-stage multiplier.

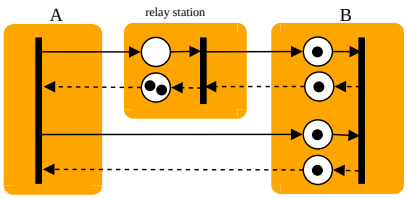

Fig. 5. Marked-graph model of the LIS of Fig. 1 with $q = 1$.

environment where it operates. The internal structure determines the maximal throughput that LIS can sustain, i.e. a LIS effectively runs with this throughput unless the environment forces it to slow down either by not providing enough valid data to process or by requiring it to wait via backpressure.[5] As discussed in the Introduction, the insertion of relay stations may change the internal structure of a LIS and have a negative impact on its performance. To quantify such impact, we define the notion of *maximum sustainable throughput (MST)* of a marked graph G as follows:

$$\theta(G) = \begin{cases} 1 & \text{if } G \text{ is acyclic;} \\ \min\left\{1, \ \frac{1}{\pi(G)}\right\} & \text{if } G \text{ is cyclic and} \\ & \quad \text{strongly connected} \\ \min_{\forall G_{SCC} \in G}\left\{\theta(G_{SCC})\right\} & \text{otherwise.} \end{cases}$$

with G_{SCC} being the component graph, where each vertex represents a strongly-connected component (SCC) of G and there is one arc between two vertices of G_{SCC} whenever there is at least one arc between the corresponding SCCs of G [19]. [6]

This definition allows us to model the impact of the LIS topology on its MST while moving from a ideal LIS with infinite queues and no backpressure to a practical LIS with finite queues and backpressure. The rationale is the following. First, since an acyclic marked graph can sustain any rate of token production/consumption, its MST is set to one by definition. Second, if G is strongly connected then its MST is equal to the reciprocal of its cycle time that is determined by any of its critical cycles. Finally, when G is cyclic with multiple SCCs then its MST is effectively determined by the slowest among them. In fact, if a slower SCC feeds a faster one then it implicitly reduces the throughput of the latter. Instead, if it is the faster SCC that is positioned up-link with respect to the slower then the LIS is not safe in terms of loss of valid data (i.e. there is unbounded token accumulation in the place of G connecting the two SCCs). In this case, we must interpret the MST as a design constraint for the LIS implementation. Since infinite queues cannot be realized, designers must satisfy this constraint by either slowing down the faster SCC or speeding up the slower. These goals may be reached explicitly by changing part of the LIS internal structure in terms of relay stations positions and shell encapsulation, but this may not always be possible. Backpressure instead provides always an implicit solution to make a practical LIS safe, but backedges introduce cycles that may lead to MST degradation with respect to the ideal LIS. In the rest of the paper we focus on how to avoid this problem.

D. The Queue Sizing Problem

To restate the problem of queue sizing: Given an ideal LIS modeled by a marked graph G with MST $\theta(G)$, after adding backpressure we have a doubled graph $d(G)$ that may have new critical cycles such that $\theta(d[G]) \leq \theta(G)$. For instance, Figure 5 shows the marked graph representation of the doubled graph in Figure 2(left) assuming $q = 1$. It is strongly-connected and the cycle $\{A, relay\ station, B, A\}$ with 3

[5]In the theoretical case where the queues are infinite, backedges may be eliminated from the model because backpressure signals are only sent when a queue is full.

[6]The *strongly connected components* (SCCs) of a directed graph are partitions of the vertices such that all vertices in an SCC are mutually reachable.

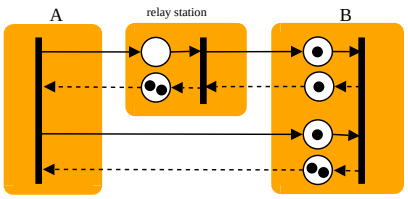

Fig. 6. Queue-sizing solution to the throughput degradation shown in Fig. 5.

places and only 2 tokens is the *critical cycle* setting the cycle time equal to $\frac{3}{2}$. Hence, the MST of this LIS is $\frac{2}{3} < 1$.

But the number of tokens in backedges can be altered by increasing the shell queues, and if enough tokens are added to the doubled graph, its MST will match the cycle time of the original "undoubled" graph. For instance, in Figure 6 the queue length for B's lower channel is increased to two so that the MST matches the one of the ideal LIS, which is equal to one. How to find the optimal queue lengths to avoid MST degradation while adding backpressure is the *Queue Sizing Problem (QS)*. In Section V we formalize QS and prove its complexity.

IV. Will fixed queue sizing work?

Fixed queue sizing is setting all queues in a system to the same given length. In the example of Figure 5, the queue sizes are set as $q = 1$. There are some classes of LISs for which fixing $q = 1$ is sufficient to maintain the optimal MST. To describe their topologies, we introduce some graph terminology. A path $p = (v_0, v_1, \ldots, v_k)$ is a sequence of vertices connected by edges and its length $|p|$ is equal to the number of its edges $(k - 1)$. A path $(v_0, v_1, \ldots, v_k)$ is simple if it has no cycles. A group of two or more simple paths is *reconvergent* if they would form a cycle if the graph were undirected. An *articulation point* is a vertex without which the graph would be disconnected [19].

A. Tree

An ideal LIS with a tree topology does not have cycles nor reconvergent paths. Fixed queue sizing is sufficient in this case because the introduction of backpressure leads to a practical LIS that is modeled by a doubled graph $d[G]$ with no cycles except those cycles between each edge and its corresponding backedge. These cycles have by construction at least two tokens. Therefore, there is no MST degradation.

B. SCC and No Reconvergent Paths

A more common, and more complicated, topology is a strongly connected component (SCC). In a special case where an SCC has no reconvergent paths, fixed queue sizing will also work.

Claim: A practical LIS whose topology is made up of SCCs with no reconvergent paths maintains the MST of the equivalent ideal LIS if it has queues of size one.

Proof: Given a graph G that is strongly connected with no reconvergent paths, let u and v be two vertices of G that are both in one of G's cycles. Since G is strongly connected, there is a path from u to v and a path from v to u along the cycle that they share. If the path from u to v is p_1 and the path from v to u is p_2, there cannot be any path from a node (not u or v) in p_1 to a node in p_2 that does not go through v. Otherwise there are reconvergent paths. Suppose there is some other vertex w in G that does not lie on the paths between u and v. There must be paths between u and w and between v and w. Without a loss of generality, suppose the path from w to u does not contain v. It must also be the case that the path from u to w does not contain v (otherwise there are reconvergent paths from w to u). From these observations, it follows that a graph G that is strongly connected with no reconvergent paths will be made up of cycles such that any vertex that belongs to more than one cycle is an articulation point (u in the discussion above). Since cycles are only connected to each other through articulation points, the

Topology	Description	Solution to MST Degradation?
Tree	No cycles, no reconvergent paths. Including DAGs with no reconvergent paths.	MST is 1. All τ's inserted by relay stations eventually leave the LIS.
SCC with no reconvergent paths	Cycles, but no reconvergent paths. To move from one cycle to another, you must pass through an articulation point.	When doubled, no new cycles will reduce the MST
Network of SCCs	Many SCCs, connected by a DAG with reconvergent paths. Two types: 1) relay stations only between SCCs, and 2) relay stations within SCCs.	Fixed queue sizing will not work in these more general graphs.

TABLE II
CLASSIFICATION OF LIS TOPOLOGIES BASED ON THEIR IMPACT ON THE THROUGHPUT DEGRADATION PROBLEM.

only new cycles (with more than two vertices) that can result from doubling G are the inverses of G's original cycles, where the *inverse* of cycle c is defined as the cycle formed by the backedges of all of c's edges. All backedges have at least one token. Thus, we are guaranteed that: (a) the inverse of cycle c has at least as many tokens as c has, and (b) the inverse does not have a smaller ratio of tokens to places than the original cycle. So the MST of the graph with backedges will not be less than the MST of the graph without backedges. Cycles between an edge and its backedge will also be added to $d[G]$, but by construction they always have two tokens. □

Likewise, a LIS with many SCCs (each without reconvergent paths) can also maintain optimal MST with $q = 1$ as long as those edges connecting its SCCs do not when doubled form a cycle that has some backedges and some forward edges - all cycles must be made of either all forward edges or all backedges. This is true when the SCCs are connected by a directed acyclic graph (DAG) with no reconvergent paths.

Table II summarizes the special cases of system topologies that we consider. Trees and SCCs with no reconvergent paths have no MST degradation as long as queues have at least size one. We prove the difficulty of optimally solving Queue Sizing for general systems in Section V, and we focus on the last case, a network of SCCs, in our experiments in Section VIII.

V. SIZING QUEUES FOR GENERAL TOPOLOGIES

While fixed queue sizing is a desirable solution, it is unfortunately only optimal for a very restricted class of topologies. In this section, we formally define the Queue Sizing Problem and show that it is NP-complete by a reduction from Vertex Cover [22]. In this discussion, when we talk about an edge or a backedge of a marked graph modeling a LIS, we mean the two arcs and the (one) place between two transitions. So, a path of length k has k places (but technically $2 * k$ arcs). Likewise, our figures will now show one arrow head per edge rather than per arc in contrast to the arcs in Figures 4-6.

Queue Sizing Problem:
Instance: A marked graph G_{QS} modeling a latency-insensitive system having MST equal to $\theta(G_{QS})$ and an integer K. Let $d[G_{QS}]$ be the doubled graph of G_{QS} where every shell has one token per place on its backedges.
Question: Is there a way to add K extra tokens to places on backedges of $d[G_{QS}]$ such that $\theta(d[G_{QS}]) = \theta(G_{QS})$ (i.e. the MST calculated before adding backedges is the same as the MST after adding backedges) ?

A. Queue Sizing ∈ NP

Checking a solution to Queue Sizing can be done with Karp's algorithm to find the maximum cycle mean before and after extra tokens are added to the graph [21], [24]. Karp's algorithm for a graph $G = (V, E)$ has complexity $O(|V||E|)$.

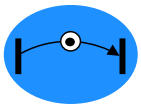

Fig. 7. Vertex construct.

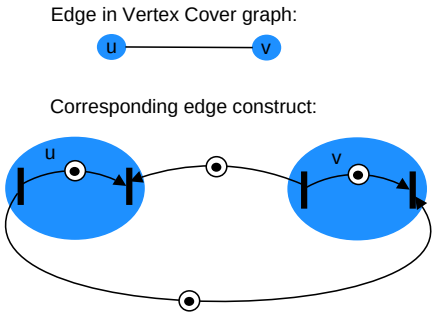

Fig. 8. Edge construct.

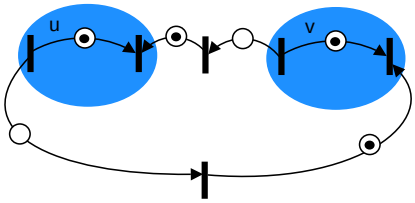

Fig. 9. Edge construct after relay stations have been added.

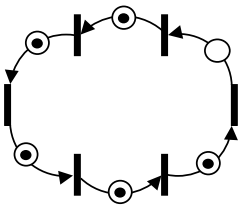

Fig. 10. A cycle that limits the ideal MST to $\frac{5}{6}$.

B. *Vertex Cover* $\propto$ *Queue Sizing*

Given an instance of Vertex Cover, a graph $G_{VC} = (V_{VC}, E_{VC})$, and an integer K, we must construct an instance of Queue Sizing, a marked graph G_{QS}, and integer K'.

1) First, for every vertex $v \in V_{VC}$, create a *vertex construct* like the one shown in Figure 7 - one edge in G_{QS};
2) Next, for every edge $(u, v) \in E_{VC}$, create a *edge construct* like the one shown in Figure 8 by adding two edges. All of the transitions in G_{QS} so far are either sources of outgoing edges or sinks of incoming edges, but not both;
3) Add relay stations to the edges added in Step 2. Figure 9 shows the resulting construct for an edge $(u, v) \in E_{VC}$;
4) Last, add a separate cycle to G_{QS} with 6 places and 5 tokens like the one in Figure 10. This addition sets the MST to $\frac{5}{6}$ since there are no other cycles in the ideal LIS.
5) Let $K' = K$.

To complete the Queue Sizing Problem instance, add in backedges as shown in Figure 11. Note that for every edge $(u, v) \in E_{VC}$, there is a cycle in G_{QS} like the one shown in Figure 12. This cycle has a mean of $\frac{4}{6} < \frac{5}{6}$, causing MST degradation. The only way to avoid this problem is to add exactly one

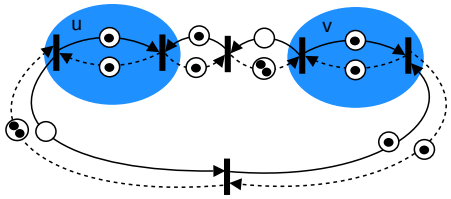

Fig. 11. Edge construct with backedges.

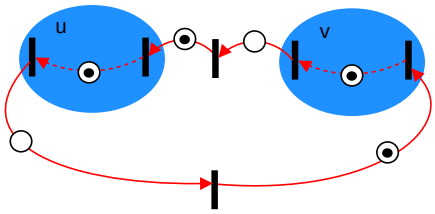

Fig. 12. Cycle in edge construct.

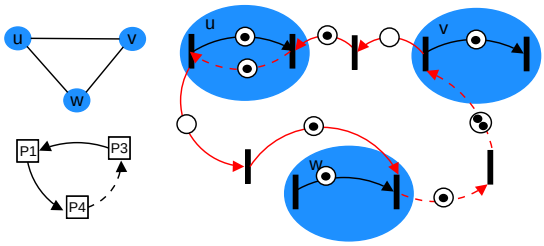

Fig. 13. Example of an additional ("side-effect") cycle.

extra token to the backedge of either the u or v vertex construct.

1) Solution to Queue Sizing $\longrightarrow$ Solution to Vertex Cover: In this step, we need to show that a solution to the Queue Sizing (QS) instance corresponds to a solution to the Vertex Cover (VC) instance. Given a solution to QS, every cycle that corresponds to an edge in the VC instance will have at least one extra token in one of the vertex constructs. Create a solution to VC instance as follows: If the vertex construct corresponding to $v \in V_{VC}$ has an extra token on its backedge, add v to the cover (i.e. the VC solution).

Since for every edge in G_{VC}, there is a corresponding cycle in G_{QS} such that one of the vertex constructs must have an extra token, every edge in G_{VC} has one endpoint in the cover. The QS solution can have only K'=K extra tokens, so the VC solution also has at most K vertices.

2) Solution to Vertex Cover $\longrightarrow$ Solution to Queue Sizing: Now assume that there is a solution to the VC instance. For every edge in G_{VC}, one of its endpoints must be in the cover. For each vertex in the cover, add one extra token to that vertex construct's backedge in G_{QS}. Then all of the cycles that correspond to edges in G_{VC} (like the ones in Figure 12) have a mean of at least $\frac{5}{6}$.

However, there are more cycles in G_{QS} than we have discussed so far. These *additional cycles* [7] are a side-effect of our edge constructs. Figure 13 shows an example of such a cycle. We must ensure that all of the additional cycles in G_{QS} have a mean greater than or equal to $\frac{5}{6}$.

We can separate each of these additional cycles in G_{QS} into two parts: parts that correspond to a vertex construct, and parts that don't. There are four ways a cycle can visit a vertex construct, shown on the left of Figure 14. Furthermore, because of the way G_{QS} is constructed, between visiting two vertex constructs, the cycle will pass through exactly two places. To help with clarity of constructing cycles, we represent these different ways of visiting vertex constructs with *P-blocks* (P is for path), shown in Figure 14. We can build a cycle by connecting P-blocks together. When putting P-blocks together, the matching edges

[7] In the previous step $QS \longrightarrow VC$, these additional cycles are already covered in the assumed QS solution.

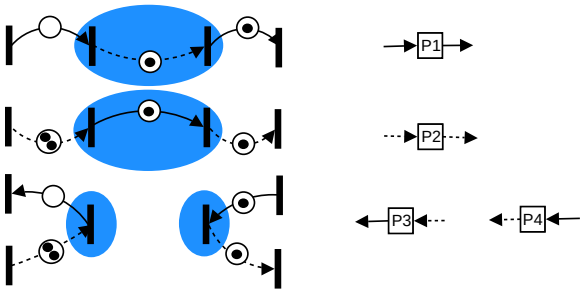

Fig. 14. The four ways to visit a vertex (left) and their P-blocks (right).

P-block	tokens	places
$P1$	2	3
$P2$	4	3
$P3$	2	2
$P4$	2	2

TABLE III
TOKENS AND PLACES PER P-BLOCK.

must both be forward or both backward (in the pictures, this means matching edges must be either both solid or both dashed). In the process of combining P-blocks, the transitions to or from which the matching edges go will be combined into one transition. For instance, P1, P4 and P3 are combined to create the additional cycle of Figure 13.

To check the mean of each additional cycle, we must take the sum of the tokens of all of its P-blocks and divide by the sum of the places of all of its P-blocks. An important observation is that given two paths, Px and Py, where $x =$ tokens(Px), $y =$ places(Px), $w =$ tokens(Py), and $z =$ places(Py); if $\frac{x}{y} \geq \frac{5}{6}$ and $\frac{w}{z} \geq \frac{5}{6}$, then $6x \geq 5y$ and $6w \geq 5z$, and so $6(x+w) = 6x + 6w \geq 5y + 6w \geq 5y + 5z = 5(y+z)$. Thus $\frac{x+w}{y+z} \geq \frac{5}{6}$. Therefore, if we break a cycle up into several paths such that each path has a path mean of at least $\frac{5}{6}$, then the cycle mean is at least $\frac{5}{6}$. By *path mean* we mean the number of tokens in the path divided by the number of places.

For each type of P-block, Table III lists its number of places and starting number of tokens, i.e. before extra tokens are added to the backedges of vertex constructs in G_{QS} according to the given solution of G_{VC}. Hence, only P1 blocks can ever have extra tokens (while all of the other P-blocks have conveniently at least as many tokens as they have places). Now, given an edge we know that one of its endpoints must be in the cover. Given a path of k vertices, where k is even, we can break the path up in to $\frac{k}{2}$ disjoint edges, and therefore we can assume that at least $\frac{k}{2}$ of the vertices are in the cover. Therefore, in a path of k P1 blocks in the QS instance, we start with a path mean of $\frac{2k}{3k}$, and then infer $\frac{k}{2}$ extra tokens, and the mean becomes $\frac{2k+\frac{k}{2}}{3k} = \frac{\frac{4k}{2}+\frac{k}{2}}{3k} = \frac{\frac{5k}{2}}{3k} = \frac{5k}{6k} = \frac{5}{6}$. Similarly, a cycle of only $P1$ blocks corresponds to a loop in the VC instance, and we know that a loop of k vertices where k is odd must have $\frac{k}{2}+1$ vertices in the vertex cover (integer division), and thus $\frac{k}{2}+1$ extra tokens in the QS graph.

Since only paths with P1 blocks can have a path mean less than $\frac{5}{6}$, we only need focus on cycles that contain P1 blocks. These cycles can be broken up into two cases:

Case 1 (Cycle of only P1 blocks). Based on our inferences above, any cycle with an even number of P1 blocks has a cycle mean equal to $\frac{5}{6}$. If the number, k, of P1 blocks is odd, where $e+1=k$, then the cycle can be broken up into two paths. The first path contains e, an even number, of P1 blocks, and so we know that there are at least $\frac{e}{2}$ extra tokens, bringing the first path's mean up to $\frac{5}{6}$. The second path contains a single P1 block, and since the loop only contains P1 blocks, we can also take into account one more "extra token", and the second path's mean is $\frac{3}{3}$. And so the overall cycle mean is $\geq \frac{5}{6}$.

Case 2 (Cycle with some P1 blocks and some other types of P-blocks). Let us break the cycle in paths of consecutive P1 blocks. In each path, there is either an odd or an even number of P1 blocks. If the

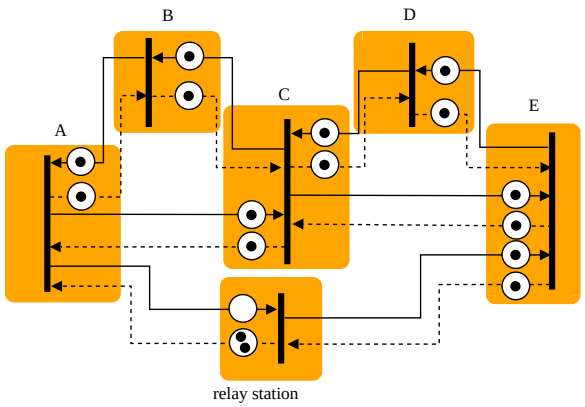

Fig. 15. An LIS where relay-station insertion is not enough.

number is even, then the path mean is $\frac{5}{6}$. If the number is odd, we can group together all but one of the P1 blocks into pairs of consecutive P1 blocks. Notice that both the incoming and outgoing edges from a path of P1 blocks are forward edges (i.e. solid edges in the figures). Since we must match forward edges to forward edges, the only way we can form a cycle by connecting a path of consecutive P1 blocks with something other than another P1 block is to "leave" the group of P1 blocks with a P4 block, and "return" to the group with a P3 block. So the P1 block that makes the odd count will always be matched by a P4 and a P3 block, and we can count those pieces together, for a total of 6 tokens and 7 places. Since the cycle can be separated into paths whose path mean is $\geq \frac{5}{6}$, the cycle mean must be $\geq \frac{5}{6}$. □

VI. Reducing MST Degradation with Relay Station Insertion

So far we have discussed queue sizing as a way to reduce MST degradation. An alternative method is to add extra relay stations to the practical LIS. This may sound counterintuitive since inserting relay stations is what causes MST degradation in the first place. In fact, relay stations can be added to a LIS for two reasons. The first is a functional reason: to break up long wire delays so that the clock rate can be reduced. The second reason is performance optimization: Casu and Macchiarulo suggest "equalizing" all reconvergent paths by inserting enough relay stations to make them have the same latency [10]. For instance, adding extra latency to one path of the LIS of Figure 2 actually increases its MST.

Inserting additional relay stations rather than increasing queue sizes has a few advantages. First, relay stations may be added anywhere along the wire, while extra logic for increasing a queue must be added within a shell (namely, the shell for which the queue holds data). This may give additional flexibility in completing the physical design of the LIS during the placement and routing phases. Furthermore, relay station insertion allows for a more modular design.

However, there are LISs where no assignment of additional relay stations can optimize performance. Figure 15 illustrates an example. Observe that the system's optimal MST is determined by the cycle $\{A$,$relay\ station$, E, D, C, B, $A\}$, whose token-to-place ratio is $\frac{5}{6}$. When backedges are considered, the cycle $\{A$, $relay\ station$, E, C, $A\}$ reduces the overall system's MST to $\frac{3}{4}$. To improve the MST using relay-station insertion, a relay station must be added to either edge (A, C) or edge (C, E). But this ends up reducing the system's optimal MST since these edges belong to small cycles. For instance, if a relay station is inserted on edge (A, C), then the cycle $\{A$, $new\ relay\ station$, C, B,$A\}$ has a token-to-place ratio of $\frac{3}{4}$.

The problem of finding an assignment of additional relay stations to optimize performance (in cases where it is possible) is NP-complete, like queue sizing. The proof is similar to the proof for queue sizing and is included in the Appendix.

Since relay station insertion cannot be used in all cases, we stick to queue sizing algorithms for our experiments.

VII. Solving the Queue Sizing Problem

Previous works have used mixed integer linear programming to solve the queue sizing problem. [26], [27]. We propose two new algorithms: a heuristic and an exact algorithm.

A. Abstraction of Queue Sizing

We first transform an instance of the Queue Sizing problem into an instance of the *Token Deficit Problem (TD)*, which is defined formally below. We do so because we want to correlate cycles that intersect with each other in the graph.

Token Deficit Problem:
Instance: Set of sets $S = (s_1, s_2, s_3, ..)$ where each $s_i \in S$ is a set $\{c_i, c_j, ...\}$ whose elements each have a non-negative deficit $d(c) \in \mathbf{Z}^*$, positive integer K.
Question: Is there a weight assignment $w(s_i) \in \mathbf{Z}^*$ to each $s_i \in S$ such that $\sum_{s_i \in S} w(s_i) \leq K$ and $\sum_{s_i \in X} w(s_i) \geq d(c_i) \ \ \forall\ c_i \in s_i$, where X is the set of all s_i such that $c_i \in s_i$?

An instance of TD is created by partitioning the cycles in the LIS marked graph of the original QS instance into sets s_i such that if $c_x, c_y \in s_i$, then c_x and c_y share edge e_i in the LIS graph. Each cycle is associated with a *deficit* equal to the number of extra tokens needed in that cycle to bring the cycle's mean above the ideal MST. This transformation abstracts away the graph structure and highlights the edges that are involved in multiple cycles. Our goal is to assign each edge a number of extra tokens such that the sum of tokens of all of a cycle's edges is greater than or equal to the cycle's deficit.

Creating an instance of TD from an instance of QS requires a list of the graph's cycles. The number of cycles is potentially exponential, though in many practical cases it is not large. We mitigate these costs by simplifying the LIS marked graphs where possible:

1) cycles whose mean is greater than or equal to the ideal MST may be ignored (including all cycles that do not have any relay stations);
2) if a set s_i is a subset of set s_j, we may omit s_i from the instance;
3) a cycle c_x that only appears in one set s_i may be automatically removed, and s_i's weight incremented by c_x's deficit;
4) if the topology of the LIS is a DAG of SCCs, possibly with reconvergent paths, but we know that relay stations are only inserted on the edges between SCCs, then we can collapse each SCC to a single vertex and work on the simplified marked graph - greatly reducing the number of cycles that must be enumerated. This particular case is discussed in more detail in Section VIII-A, and we show in Section VIII-C that our heuristic algorithm performs well for larger graphs of this type.

Observe that there always exists a number K for which TD can be solved [26]. An easy way to look at this is to consider that every relay station introduces one void data item, or τ, into the LIS, and if there are R relay stations, no cycle can be deficient in more than R tokens. Hence, adding R extra tokens to one edge in each cycle that has backedges guarantees that none of them will have a cycle mean less than one.

The Token Deficit Problem is also NP-complete. This can be shown with a reduction from Dominating Set. We do not include the proof, but instead refer the reader to [17].

B. Algorithms

We propose a heuristic algorithm that produces a solution in $0(|S|^2|V||C|)$ time, where $|C|$ is the number of cycles and $|V|$ is the number of vertices in the original LIS graph. For comparison purposes, we also develop an algorithm that produces the optimal solutions for the TD problem.

Heuristic Algorithm: Given an instance of the Token Deficit Problem, assign to each element $s_i \in S$ a weight equal to the maximal deficit among its elements. By construction, this initial assignment is a solution. Now,

1) for each $s_i \in S$ whose weight is not yet fixed, decrement $w(s_i)$ and check that the weight assignment is still a solution. If it is a solution, leave the new weight of s_i, if not increment and fix $w(s_i)$ back to its value at the beginning of the step;
2) repeat Step 1 if any $w(s_i)$ is unfixed. Otherwise, stop.

To check that the weight assignment is correct costs $\bigcirc(|S||C|)$, and $\sum_{s_i \in S} w(s_i)$ can be at most $|S||V|$, therefore the overall complexity of this algorithm is $\bigcirc(|S|^2|V||C|)$.

Exact Algorithm: First, the graph instance is expanded by replicating the sets s_x so that if D is the largest deficit of the elements of s_i, then s_i will be replicated D times. This simplifies the problem since for all weights, $w(s_x) \in \{0, 1\}$. Then, we perform a binary search on K whose values vary from $K = 1$ to $K = the\ heuristic\ solution$. For each round of the binary search, we build a K-depth search tree that branches by choosing one of the edges to have $w(s_x) = 1$. In the worst case (a "no" answer), the search tree takes $O((|S|D)^K)$ time.

VIII. Experimental Analysis

We evaluated our heuristic algorithm completing a set of experiments with LISs that were derived through random graph generation. We built a *graph generator* that takes as inputs: v (number of vertices), s (number of strongly-connected components), c (minimum number of cycles within each SCC), rs (number of relay stations), whether or not reconvergent paths are allowed between SCCs (rp = 1 for yes, 0 for no) and a policy for relay-station insertion (either any or scc). Graphs are generated with the following steps:

1) partition the graph into SCCs;
2) for each SCC s:
   a) make a cycle that visits all of the vertices in s;
   b) choose $u, v \in s$ such that (u, v) is not an edge of s and add (u, v) to s;
   c) repeat Step 2b c times; this guarantees that at least c cycles are added to s as long as there are enough possible edges in s so that an unused (u, v) can always be chosen;
3) create a connected auxiliary graph H whose vertices correspond to SCCs in the generated graph and whose edges are randomly chosen avoiding to create cycles between SCCs (reconvergent paths are allowed if $rp = 1$);
4) for each edge (s_1, s_2) between SCCs s_1 and s_2 in H choose vertices $v_{s_1} \in s_1$ and $v_{s_2} \in s_2$, and add edge (v_{s_1}, v_{s_2}) to the graph;
5) insert relay stations randomly on edges that satisfy the chosen policy: with policy any they may be inserted on any edge while with policy scc they may be inserted only on edges that connect SCCs; i.e., those edges added in Step 4.

The results presented below are the average of 50 trials where graph topology and the specific locations of relay stations are selected randomly.

A. MST Degradation

Backpressure causes a degradation of maximal sustainable throughput in cases where (1) a graph contains a cycle that is made up of both backedges and forward edges, (2) one or more of the forward edges in the cycle has had relay station insertions, and (3) where there are more relay stations than the amount of extra queue space on the backedges. Figure 16 contrasts the change in MST when we move from infinite to finite queues. Clearly to make topology restrictions on where relay stations may be inserted has a large impact on MST. When relay stations are restricted to edges between SCCs (scc insertion), the MST with infinite queues is optimum at 1.0. The MST over finite size queues ($q = 1$) for scc insertion does degrade between 15% and 30%; however, it is still significantly higher than the MST when relay stations can be inserted within SCCs, no matter how large the queues are. When relay stations are inserted anywhere in the graph (any insertion), there is not much difference in MST as the queue sizes increase.

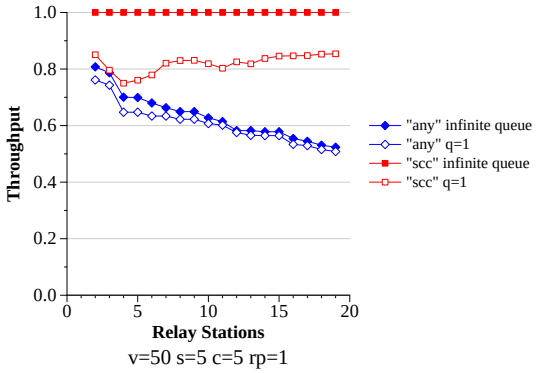

Fig. 16. MST of graph (v=50,s=5,c=5,rp=1) given infinite and finite ($q = 1$) queues.

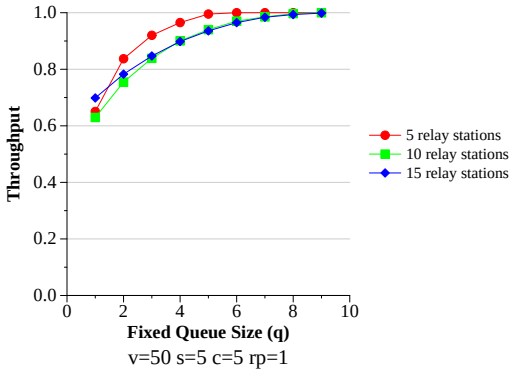

Fig. 17. MST Improvement using Fixed Queues.

This is simply because new cycles introduced in the graph when backedges are considered usually do not introduce lower token-to-place ratios than the cycles without backedges. In the case of scc insertion, there are no cycles with relay stations until after the backedges are added into consideration. In the rest of the paper, we will focus on graphs that use scc insertion since this is where the most improvement is needed.

B. Fixed-Size Queues

Figure 2 is an example of LIS where optimal MST cannot be maintained with $q = 1$. There is no fixed queue size that will provide optimal MST in arbitrary graph topologies. To construct a LIS that does not have optimal MST with fixed queues of size q, take Figure 2 and add $(q - 1)$ more relay stations to the upper channel between A and B. In extreme cases, fixed queue sizing will not work; however, in average and typical cases, fixing the sizes of the queues to the same value can be a fast and effective approach. Figure 17 shows the MST improvements that are gained in LIS derived with our graph generator as the fixed queue size q increases. On average, with $q = 1$, the MST can be as low as 65% of the optimal, but when $q >= 5$, the MST is above 90% of the optimal.

C. Exact vs. Heuristic Solution

Table IV lists experimental results using LISs with the following topology: SCCs connected with reconvergent paths, where ten relay stations are inserted only on the edges between SCCs. This topology allows us to use some optimization steps to greatly reduce the graph size before adjusting queue sizes. Since no relay stations are added within SCCs and there are no cycles between SCCs, any cycle that degrades the MST after backpressure is added must have inter-SCC backedges. So we can optimize the

(V,E)	# SCC	# Edges (inter-SCC)	Cycles (inter-SCC)	RS	Exact Soln.	Heuristic Soln.	% Exact finished	# Cycles in Unfinished	Heuristic Soln. - no Exact
(50,82.00)	10	12.00	26.25	10	3.44	3.69	0.96	245.00	10.50
(100,122.06)	10	12.06	41.15	10	3.48	3.65	0.96	328.00	9.00
(100,144.71)	20	24.71	171.14	10	3.79	4.07	0.56	32032.09	9.73
(200,222.10)	10	12.10	40.76	10	3.20	3.31	0.98	802.00	8.00

TABLE IV
HOW GOOD ARE THE SOLUTIONS RETURNED BY THE HEURISTIC ALGORITHMS?

MST by adding tokens to the inter-SCC edges only. Also, since there are no cycles with relay stations and without backedges, we know that the optimal MST is equal to 1. This means that we simply need to add extra queue tokens to the backedges so that every cycle has at least as many tokens as places. With these observations, we can collapse the SCCs to single nodes and solve the queue-sizing problem considering only the inter-SCC edges and far fewer cycles.

Each experiment shows the average values over 50 different graphs. '*(V,E)*' gives a characterization of the graph in terms of the number of vertices and edges. '*# Edges (inter-SCC)*' is the average number of edges between SCC. '*Cycles (inter-SCC)*' is the average number of cycles between SCCs (after backedges have been added). '*RS*' is the number of relay stations added to the system. As mentioned at the end of Section VIII-A, these experiments do not put relay stations within an SCC (only between SCCs). '*Exact Soln.*' lists the average amount of additional queue space (number of tokens added to the marked graph representation) that is necessary to optimize performance using the exact algorithm. '*Heuristic Soln.*' shows the average amount of queue space needed when using the heuristic. In some cases, the exact program was halted after running for more than an hour. '*% Exact finished*' refers to the percent of 50 trials that it completed in under an hour. For these cases '*# Cycles in Unfinished*' and '*Heuristic Soln - no Exact*' tell the number of cycles and the heuristic solution.

The heuristic performs very well in these experiments, producing solutions within 8% of the exact algorithm in every case. In addition, it can handle much larger problems. One limitation is that the initial listing of all the cycles, a necessary step in the heuristic algorithm, may blow up fairly quickly. Using our topology-based optimization of collapsing SCCs, the number of vertices can actually scale much higher than the experiments shown here, provided that the number of SCCs remains relatively low and it is possible to only add relay stations between SCCs.

IX. CONCLUDING REMARKS

Back-pressure is a logical mechanism to control the flow of information on a communication channel and guarantee that no data is lost. Adding backpressure to a latency-insensitive system (LIS), however, can cause a degradation of its maximal sustainable throughput (MST). This degradation can be corrected by increasing the shell queues on communication channels that are a bottleneck for performance and/or by inserting relay stations along channels that have some slack. We studied how the LIS topology impacts the MST degradation and how it is related to the different solutions. When a LIS is made up of SCCs with no reconvergent paths, or a tree of SCCs with no reconvergent paths, using fixed-size queues achieves optimal MST. In more general topologies, using relatively small fixed-size queues can often bring performance within 90% of the optimal MST. However, we also show that the queue sizing problem for optimal MST is NP-complete. This motivated us to develop a heuristic that produces solutions that are close to the exact one while being able to handle much larger problems. Interestingly enough, in our experiments the class of graphs with the greatest MST degradation, i.e. the class of directed acyclic graphs of SCCs that only have relay stations between SCCs, can be easily simplified with a straightforward optimization.

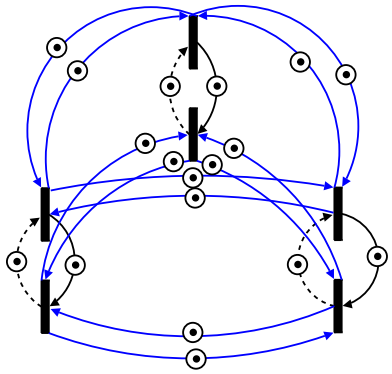

Fig. 18. Vertex construct for RSI problem.

APPENDIX

Relay Station Insertion Problem:
Instance: A Marked Graph modeling a Latency-Insensitive System, as described in Section III, and an integer K.
Question: Is there a way to add at most K Relay Stations to the graph such that the doubled graph has ideal maximal sustainable througput?

The difference in this problem compared to Queue Sizing is that while we can insert a Relay Station to balance throughput (since the backedge of a relay station has additional queue capacity) the relay station also increases the latency along the path where it is added. This additional latency may also reduce the throughput of cycles in the system. There exist systems where the throughput cannot be improved to the ideal MST using only relay station insertion.

A. Relay Station Inertion $\in$ NP

As with the Queue Sizing problem, a potential Relay Station Insertion solution can be checked with Karp's algorithm.

B. Vertex Cover $\propto$ Relay Station Insertion

Given an instance of Vertex Cover, a graph $G_{VC} = (V_{VC}, E_{VC})$ and an integer K_{VC}, we must construct an instance of the Relay Station Insertion Problem.

Since Vertex Cover is NP-complete even for planar graphs with degree at most three [22], we assume that no vertex has degree more than three.

Construct an instance of Relay Station Insertion as follows:

1) Let the integer $K_{RSI} = 3 * K_{VC}$;
2) for each vertex in V_{VC}, create a vertex construct like the one in figure 18. Each construct starts with three backedges, and a total of six transitions. Three of the transitions are the source of the backedges, and three are the sinks/destinations. We connect these transitions by adding all possible edges between the three source transitions and all posible edges between the sink transitions (twelve edges - shown in blue in the Figure 18). In the discussion below, we do not consider the backedges of these edges, because since the blue edges are added in both directions, there is no backedge that causes a new cycle in the doubled graph;
3) For each edge $(u, v) \in E_{VC}$, we add edges between two vertex constructs for u and v similarly to how we added edges for the Queue Sizing edge constructs, two edges to make a cycle with a backedges from each vertex construct, and adding a relay station to both inter-vertex paths. However, this time we have three backedges to choose from in the vertex constructs. For each incoming edge to a vertex construct, connect it to a different backedge. Since we've assumed that there are not

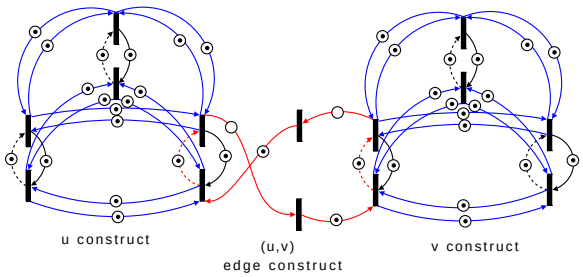

Fig. 19. Edge construct for RSI problem.

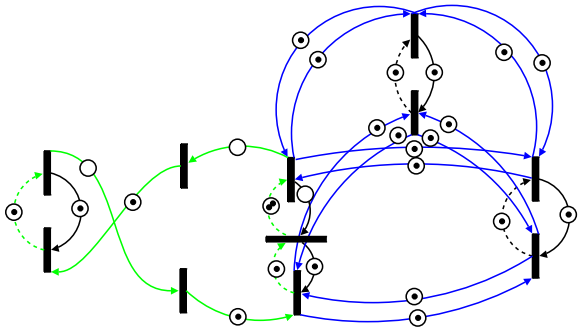

Fig. 20. Adding a Relay Station to the Edge construct.

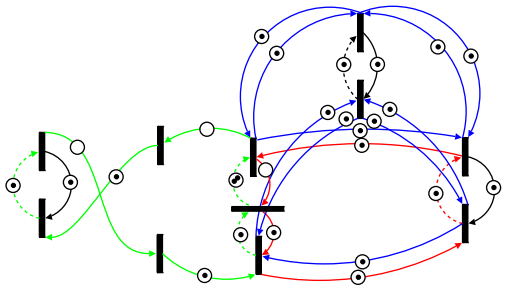

Fig. 21. Adding a Relay Station to the Edge construct breaks another cycle...

more than three incoming edges, each incoming edge will have a distinct backedge. In the case where there are fewer than three incoming edges, we leave the extra backedges unconnected - they are "dummy" edges, but we need them later to maintain an equal cost of adding a vertex of high degree and of low degree to the Vertex Cover;

4) As with Queue Sizing, again construct an extra cycle that limits the MST to $\frac{5}{6}$.

1) Solution to Relay Station Insertion $\longrightarrow$ Solution to Vertex Cover: Every edge in E_{VC} corresponds to a cycle in G_{RSI} with a mean of $\frac{4}{6}$, as shown in Figure 19. Given a Solution to Relay Station Insertion, we know that a relay station must have been inserted in one of the vertex constructs, as in Figure 20. But this will create latency in the cycles within the vertex construct (Figure 21). Therefore, if a relay station is inserted to a vertex construct to increase the storage capacity of one backedge, then two more relay stations must be inserted for the other two backedges (figure 22). So, for every edge in E_{VC}, at least one vertex construct will have had three relay stations added. If the RSI solution uses $3K$ or fewer extra relay stations, then it must correspond to a VC solution with K or fewer vertices in the Cover.

2) Solution to Vertex Cover $\longrightarrow$ Solution to Relay Station Insertion: Given a solution to Vertex Cover, for each vertex in the Cover, add three relay stations to that vertex's construct in the RSI instance, like in figure 22. This will take care of the cycles constructed explicitly from the edges of G_{VC}.

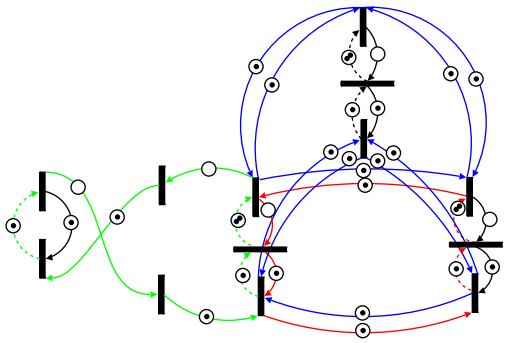

Fig. 22. Add 3 Relay Stations to really fix the cycle.

We will reason about the extra cycles that can be formed in G_{RSI} using *P-blocks* similar to those in Figure 14 for the Queue Sizing construction. Consider Figure 23. It enumerates all possible ways to visit a vertex construct. Let us refer to the different cases as P1, P2, etc. Note that for P1-P4 and P1'-P4', we enter and leave through the transitions that are connected to the same backedge in the vertex construct. This means that there is only one way that the incoming and outgoing edges can go - they both go to the same vertex construct, and the cycle will visit exactly two vertex constructs.

We can reduce most of these cases. Relay Stations will be added to only three of the edges in each construct (this is by our construction from the first line of this subsection). If two P-blocks have the same incoming and outgoing edges and the same number of edges that *could* have had relay stations added (or backedges of edges that could have relay station added), then we can reduce one to the other. Call these edges of interest "interesting edges". Consider P1 and P3. They both have the same number of interesting backedges, but P3 has two extra inter-vertex construct edges. Using the same observation that we can find a lower bound of a cycle's mean based on lower bound of all of the path means of paths that make up the cycle, we can reduce P3 to P1 since any cycle that includes P3 corresponds to a cycle that corresponds P1 and the cycle with P1 will have a lower cycle mean.

We can also reduce P8 to P5. In this case, P5 has no interesting edges, but P8 has one interesting forward edge and one interesting backedge. But since Relay Stations will either be added to all interesting forward edges or none, those on the two interesting edges in P8 will cancel each other out.

Using these reductions, we can reduce all of the cases to P1, P1', P5, P5', P11, and P11'. P1 and P1' can only make two types of cycles: Cycle Type 1: P1,P1; Cycle Type 2: P1',P1'. Cycle Type 1 corresponds to the cycle explicitly constructed for each edge, and have already been discussed. It is easy to see that cycles with Cycle Type 2 will also have a mean greater than $\frac{5}{6}$.

Of the remaining P5,P5',P11 and P11', only P11 can have a path mean less than $\frac{5}{6}$. (a relay station could be added to P11' interesting edge, but then P11 will have a mean of $\frac{5}{5} > \frac{5}{6}$). If there are two P11 blocks next to each other, they correspond to an edge in G_{VC}, so one of them must have had a relay station added to it's interesting edge (that is, the forward edge corresponding to it's interesting edge). Therefore the overall path mean of the two P11 blocks is at least $\frac{8}{9} > \frac{5}{6}$. If there are three P11 blocks in a row, then one must have a relay station, so the overall path mean is at least $\frac{11}{13} > \frac{5}{6}$. If a P11 block is not next to another P11 block, the only other block it could connect to is a P5 block, and the path mean of the P11,P5 path would be $\frac{6}{7} > \frac{5}{6}$. For any cycle, the cycle mean will always be greater than $\frac{5}{6}$, and the solution to Vertex Cover corresponds to a solution to Relay Station Insertion. □

ACKNOWLEDGMENTS

This work has been partially sponsored by an NDSEG fellowship and the GSRC Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

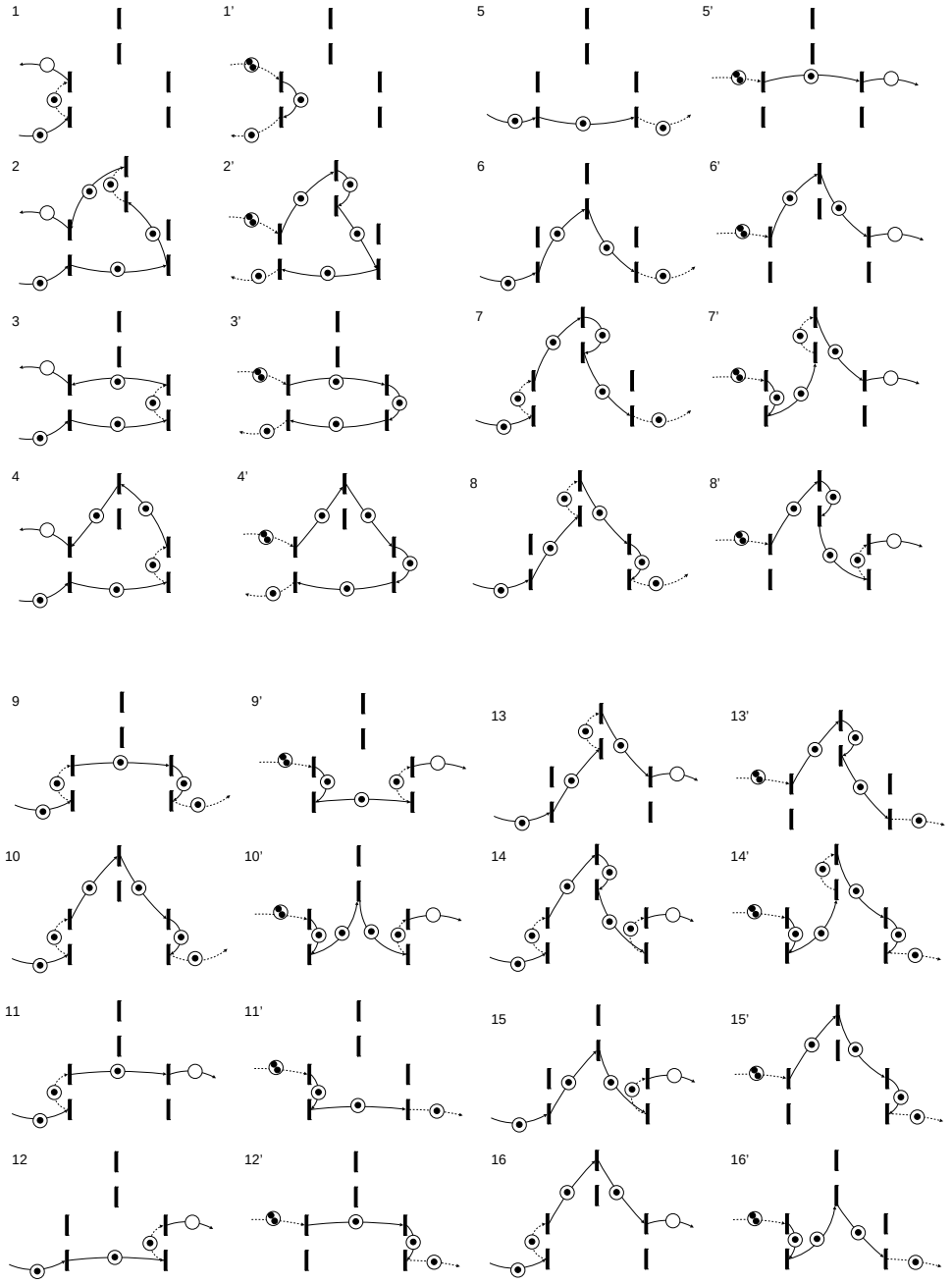

Fig. 23. Ways to visit vertex constructs from cycle in the RSI construction.

REFERENCES

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

[2] A. O. Balkan, M. N. Horak, G. Qu, and U. Vishkin. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In *IEEE Symposium on High-Performance Interconnects*, Aug. 2007.

[3] A. O. Balkan, G. Qu, and U. Vishkin. A mesh-of-trees interconnection network for single-chip parallel processing. In *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, pages 73–80, 2006.

[4] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, Jan. 2003.

[5] S. M. Burns. *Performance analysis and optimization of asynchronous circuits*. PhD thesis, California Institute of Technology, Pasadena, CA, USA, 1991.

[6] 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, San Jose, CA, Nov. 1999. IEEE.

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

[8] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Proc. of the Design Automation Conf.*, pages 361–367, 2000.

[9] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Coping with latency in SOC design. *IEEE Micro*, 22(5):24–35, Sep-Oct 2002.

[10] M. R. Casu and L. Macchiarulo. Issues in implementing latency insensitive protocols. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 1390–1391. IEEE, 2004.

[11] M. R. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *Proc. of the Design Automation Conf.*, pages 576–581, 2004.

[12] M. R. Casu and L. Macchiarulo. Throughput-driven floorplanning with wire pipelining. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(5):663–675, May 2005.

[13] V. Chandra, H. Schmit, A. Xu, and L. Pileggi. A power aware system level interconnect design methodology for latency-insensitive systems. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 275–282, 2004.

[14] V. Chandra, A. Xu, H. Schmit, and L. Pileggi. An interconnect channel design methodology for high performance integrated circuits. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 21138–21143, 2004.

[15] P. Cocchini. Concurrent flip-flop and repeater insertion for high-performance integrated circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 268–273, 2002.

[16] R. L. Collins and L. P. Carloni. Topology-based optimization of maximal sustainable throughput in a latency-insensitive system. In *Proc. of the Design Automation Conf.*, pages 410–415, 2007.

[17] R. L. Collins and L. P. Carloni. Topology-based optimization of maximal sustainable throughput in a latency-insensitive system. Technical Report CUCS-008-07, Columbia University, New York, New York, Feb. 2007.

[18] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *J. Comput. Syst. Sci.*, 5(5):511–523, 1971.

[19] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 2001.

[20] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. $\times$*pipes*: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs. In *Proc. Intl. Conf. on Computer Design*, pages 536–541, Oct. 2003.

[21] A. Dasdan and R. K. Gupta. Faster maximum and minimum mean cycle algorithms for system-perofrmance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, 1998.

[22] M. R. Garey and D. S. Johnson. The rectilinear Steiner tree problem is np-complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977.

[23] S. Hassoun and C. J. Alpert. Optimal path routing in single and multiple clock domain systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(11):1580–1588, Nov. 2003.

[24] R. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics, Discrete Math*, 23(3):309–311, September 1978.

[25] S. Kim and P. A. Beerel. Pipeline optimization for asynchronous circuits: complexity analysis and an efficient optimal algorithm. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 296–302, 2000.

[26] R. Lu and C.-K. Koh. Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 227–231, 2003.

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

[28] R. Lu, G. Zhong, C.-K. Koh, and J.-Y. Chao. Flip-flop and repeater insertion for early interconnect planning. In *Proc. of the Conf. on Design, Automation and Test in Europe*, Mar. 2002.

[29] R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *MPC '98: Proceedings of the Mathematics of Program Construction*, pages 272–285, London, UK, 1998. Springer-Verlag.

[30] T. Murata. Circuit theoretic analysis and synthesis of marked graphs. *IEEE Transactions on Circuit and Systems*, 24(7), July 1977.

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

[32] P. Pan, A. Karandikar, and C. L. Liu. Optimal clock period clustering for sequential circuits with retiming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(6):489–498, 1998.

[33] P. Prakash and A. J. Martin. Slack matching quasi delay-insensitive circuits. In *Proc. of the IEEE Intl. Symposium on Asynchronous Systems and Circuits*, March 2006.

[34] 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, Sept. 1980.

[35] R. Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):309–311, Oct. 1968.

[36] L. Scheffer. Methodologies and tools for pipelined on-chip interconnect. In *Proc. Intl. Conf. on Computer Design*, pages 152–157, Oct. 2002.

[37] T. Yamada and S. Kataoka. On some LP problems for performance evaluation of marked graphs. *IEEE Transactions on Automatic Control*, 39(3):696–698, 1994.