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 (SoC) engineers to decouple the design of the computing cores from the design of the intercore communication channels while following the synchronous design paradigm. In a latency-insensitive system (LIS), each core is encapsulated within a shell, which is a synthesized interface module that dynamically controls its operation. At each clock period, if new data have not arrived on an input channel or if 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 (QS). We evaluate the heuristic algorithm with experiments over a large set of synthetically generated systems and with a case study of a real SoC system. We find that the topology of a LIS can impact not only how much throughput degradation will occur but also the difficulty of finding optimal QS solutions.**

***Index Terms*—Latency-insensitive design (LID), performance analysis, system-level design, systems-on-chip (SoCs).**

I. Introduction

LATENCY-INSENSITIVE design (LID) [7], [8] is a correct-by-construction methodology for system-on-chip (SoC) design 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 characterizes nanometer technologies [10]. In particular, LID decouples the design of the IP cores from the design of the communication channels among them. Moreover, for the latter, it eases the application of *wire pipelining*, which is a technique to fix timing violations in global interconnect that is both effective and challenging [2], [18], [47].

Given a netlist of IP cores, which may be specified as synthesizable register-transfer level (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 intercore 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, e.g., through *clock gating*. Intershell channels made of long wires can be pipelined through the insertion of *relay stations* (clocked buffers with twofold storage capacity [7]) 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 redesign of any IP core and without jeopardizing the system behavior [8]. Essentially, this is possible because of the following: 1) The data exchanged by the shells are marked as either valid[1] or void; 2) the relay stations are initialized with void data; and 3) 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 shell fires the core if and only if it has new valid data from each input channel, and it stalls the core if 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 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 [8].

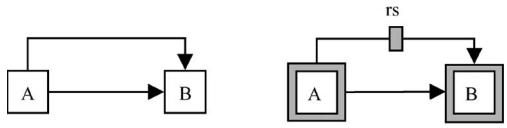


Fig. 1. Example of a system transformed into a LIS. A and B are encapsulated in shells, and a relay station is inserted on the upper channel of A.

The simple example in Fig. 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 us 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. 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

Manuscript received January 28, 2008; revised May 6, 2008 and July 15, 2008. Current version published November 19, 2008. This work was supported in part by an NDSEG fellowship and in part by the GSRC Focus Center, which is one of the five research centers funded under the Focus Center Research Program, which is a Semiconductor Research Corporation program. This paper was recommended by Associate Editor G. E. Martin.

The authors are with the Department of Computer Science, Columbia University, New York, NY 10027 USA (e-mail: rlc2119@cs.columbia.edu; luca@cs.columbia.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2008.2008914

[1]Valid and void data are also denoted as informative and stalling events in the theory of latency-insensitive design, respectively [8].

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

output channel	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	...

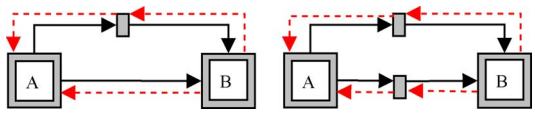


Fig. 2. (Left) Adding backedges to the LIS example. (Right) Inserting an additional relay station for performance reasons.

latched output is initialized to zero. We use τ to denote a void data item, as proposed in [8].

Aside from the "traditional" clock frequency of its synchronous circuits, the main performance metric of a LIS is the rate at which it processes valid data [9]. This throughput, which may be reduced by the periodic occurrence of void data, depends on the following 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, which is the rate at which the LIS effectively processes valid data 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 [9], [36]. 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 in Fig. 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., $\theta = \#\ valid\ data\ items/clock\ periods = 1$. 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 one) in the corresponding input queue while waiting for the first valid data on A's upper channel (equal to zero) 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 3/4 feeds another downlink subsystem with a lower MST of 2/3, only the presence of queues of infinite size (*infinite queues*) could provide the shells of the latter with sufficient buffering capacity. However, 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* [8], guarantees lossless communication. However, 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 Fig. 2, 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 backward edges. Now, if we suppose that the shells have queues with fixed capacity $q = 1$, the MST of the system on the left of Fig. 2 becomes 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 the 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 unrealizable) 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 the MST of G, and $d[G]$ is the marked graph obtained by adding backedges to G (the *doubled graph* of G). It has been shown that $\theta(d[G]) = \theta(G)$ when the system has finite queues that are "big enough" [36]. 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 (QS) 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 in Fig. 2, 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.

In this paper, 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 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 QS 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 QS with the alternative method of relay-station insertion, we demonstrate that the latter has more limited applicability by presenting the counterexample of a LIS whose MST cannot be optimized by only adding relay stations (Section VI). Finally, we propose a heuristic algorithm for the

QS problem (Section VII) and evaluate empirically how well it performs compared with an exact algorithm (Section VIII). We observe that, in the most difficult class of topologies, instances with the greatest throughput degradation are typically very amenable to simplifications.

The main contributions of this paper include a proof of the complexity of optimal QS, a characterization of topologies that maintain optimal throughput with fixed-size queues, and a heuristic for sizing queues, which produces solutions close to optimal in a fraction of the time.

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 [7], [18], [28], [37], [42], [47]. The 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], [14], [15], [24].

The performance analysis of LISs originally presented in [9] 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 [35], [36]. In order to find optimal queue sizes, they employ mixed integer linear programming (MILP).

Casu and Macchiarulo avoid QS issues by scheduling the core firing and eliminating backpressure [12], [13]. 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 [11] 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 in Fig. 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 QS and relay-station insertion problems for LISs. The first problem is related to buffer sizing optimization in synchronous data flows (SDFs), which is an important step in software synthesis for both single-appearance scheduling on a single digital signal processor [5] and deadlock-free scheduling on multiprocessor architectures [27], [48]. Poplavko *et al.* use SDFs for reasoning on the buffer sizing of the channels of a network-on-chip (NoC) to optimize the performance of a multiprocessor architecture [43]. Hu *et al.* propose an efficient greedy algorithm to size the input queues in a NoC router given the application traffic characteristics such that the NoC performance is maximized while satisfying a total buffering resource budget [29]. Maxiaguine *et al.* present a mathematical framework for the performance analysis of streaming applications once the on-chip buffer constraints are given [39]. The problems that we address in this paper are different from the ones in these works due to the particular constraints imposed by the AND-*firing* policy of the shells and the distinct buffering roles that relay stations and shell queues play in LID.

Our approach is somewhat more related to the slack matching problem that has been defined for quasi delay-insensitive *asynchronous* systems [38], [44]. 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 QS, however, we do not break up core/shell pairs, but we simply add extra storage capacity on the backpressure paths. Moreover, with relay-station insertion, we pipeline wires between computational cores but not the core logic itself. Venkataramani and Goldstein [49] approach slack matching similarly by inserting buffers along channels. However, this differs from our definition of QS since buffers introduce additional latency along the backward paths, while our queues do not. In our discussion of relay-station insertion in Section VI, we show that there exists a graph where this addition of latency prevents us from optimally solving the system's throughput degradation. The slack matching problem has been modeled with marked graphs and proven NP-complete by Kim and Beerel [31] and has been solved with algorithms that are based on MILP by Prakash and Martin [44].

In this paper, we forgo the popular MILP 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 [19] on throughput degradation in LISs with a proof about the complexity of optimal QS.

III. Modeling a LIS With Marked Graphs

Marked graphs, also known as *decision-free* Petri nets, are a simple model for concurrent systems [22] 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 of the following: 1) They operate as deterministic systems and 2) 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, and $M_0 : P \to \mathbf{Z}^*$ is the initial marking (or state), such that $P \cap T = \emptyset \wedge P \cup T \neq \emptyset$ and $\forall p \in P(|\{t|(t,p) \in F\}| = |\{t|(p,t) \in F\}| = 1)$.

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 [22]. 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], [22], [40], [41].

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 eventually 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 time* $\pi(t)$ of a transition t of G is the average time separation between two consecutive firings of t and 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 [45].[2] 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. The cycle time can be computed using Karp's algorithm to find the minimum cycle mean of a directed graph [25], [30] or using linear programming [6], [50].

B. Modeling LISs With Marked Graphs

We define the *cycle mean* of a cycle c of G as the ratio of the number $M_0(c)$ of tokens that are present on c divided by the sum of the delays of its transitions, i.e.,

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

As mentioned earlier, the cycle time $\pi(G)$ of G is equal to the reciprocal of the minimum cycle mean across all cycles in G. Cycles whose cycle mean coincides with $\pi(G)$ are called *critical cycles*.

Since LISs are synchronous systems, we model them using timed marked graphs such that $\forall t \in G(d(t) = 1)$. Hence, the denominator of the cycle mean of a cycle c coincides with the number of transitions in the cycle (which is equal to the number of places); in turn, the cycle mean becomes equal to the ratio of places and tokens around the cycle.

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 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 *time stamp* or clock period).

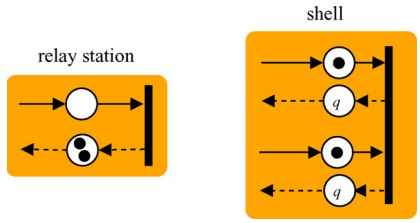


Fig. 3. Marked-graph models of relay stations and shells with backpressure.

Fig. 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 edge models 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 edge has no token since it must produce a τ in the first time stamp 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 time stamp, and its backedges have a number q of tokens that are equal to the capacity of the corresponding input queue.

Fig. 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 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 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 [7]. Moreover, a shell relies on the logic of its stallable core to latch the output signals and features bypassable 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.[3] 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: 1) If a transition has an incoming place with one token, then that transition corresponds to a shell in the

[2]Similar results are found in [6], [40], and [46].

[3]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.

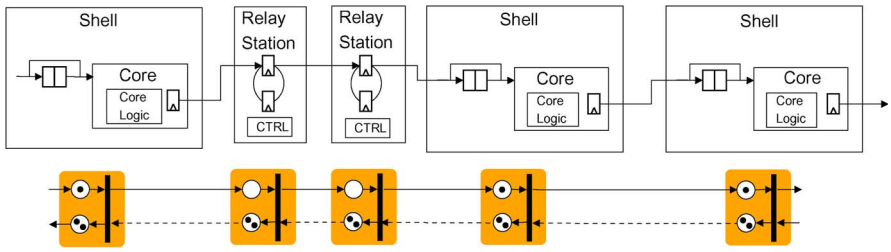


Fig. 4. Marked-graph model (with $q = 2$) of a path across multiple shells and relay stations in a LIS.

LIS, and all of its incoming places must have one token; and 2) 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. Moreover, 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 environment where it operates. The internal structure determines the maximal throughput that the 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.[4] As discussed in the introduction, the insertion of relay stations may change the internal structure of a LIS and has a negative impact on its performance. To quantify such impact, we define the notion of *maximal sustainable throughput (MST)* of a marked graph G as follows:

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

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 [23].[5]

This definition allows us to model the impact of the LIS topology on its MST while moving from an 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 uplink 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.[6] These goals may be reached explicitly by changing part of the LIS internal structure in terms of relay-station positions and shell encapsulation, but this may not always be possible. Backpressure always provides 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 this paper, we focus on how to avoid this problem.

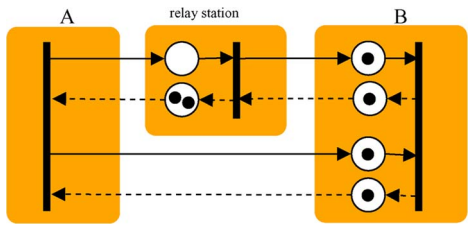


Fig. 5. Marked-graph model of the LIS in Fig. 1 with $q = 1$.

D. 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, Fig. 5 shows the marked-graph representation of the doubled graph in Fig. 2, assuming $q = 1$. It is strongly connected, and the cycle $\{A, relay\ station, B, A\}$ with three places and only two tokens is the *critical cycle* setting the cycle time equal to 3/2. Hence, the MST of this LIS is $2/3 < 1$.

However, 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 Fig. 6, the queue length for the lower channel of B is increased to two so that

[4]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.

[5]The SCCs of a directed graph are partitions of the vertices such that all vertices in an SCC are mutually reachable.

[6]The problem of interfacing SCCs operating at different throughput values will arise also for globally asynchronous locally synchronous (GALS) systems [16], which are considered an interesting alternative to designing large clock trees for billion-transistor chips. GALS systems are made of synchronous clusters, possibly running at different clock frequencies, that are connected by asynchronous interconnection networks. They will require mixed-timing interface circuits like those proposed in [17].

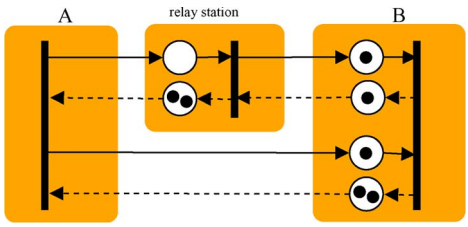


Fig. 6. QS solution to the throughput degradation shown in Fig. 5.

the MST matches the ideal value, which is equal to one. How to find the optimal queue lengths to avoid MST degradation while adding backpressure is the *queue sizing (QS) problem*. In Section V, we formalize QS and prove its complexity.

IV. Will Fixed QS Work?

Fixed QS is setting uniformly all queues in a system to the same given length. In the example in Fig. 5, the queue sizes are set as $q = 1$. There are some classes of LISs for which fixing q to the same size for all shells is sufficient to maintain the ideal MST regardless of the number of relay stations added to the system. 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). 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 was undirected. An *articulation point* is a vertex without which the graph would be disconnected [23].

A. Tree

An ideal LIS with a tree topology does not have cycles or reconvergent paths. Fixing the queue size to one 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 complicated, topology is an SCC. In a special case where an SCC has no reconvergent paths, fixed QS also works.

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, which are both in one of the cycles of G. 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 that 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 that 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 earlier). Since cycles are only connected to each other through articulation points, the only new cycles (with more than two vertices) that can result from doubling G are the inverses of the original cycles of G, where the *inverse* of cycle c is defined as the cycle formed by the backedges of all of the edges of c. All backedges have at least one token. Thus, we are guaranteed the following: 1) the inverse of cycle c has at least as many tokens as c has and 2) the inverse does not have a smaller ratio of tokens to places than the original cycle. Thus, 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 from a cycle that has some backedges and 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 can be guaranteed to have no MST degradation using fixed QS. In fact, no extra queue space $(q = 1)$ is needed to make this guarantee. For other topologies, fixed QS with $q = r + 1$, with r being the total number of relay stations that are present in the LIS, is sufficient to maintain the ideal MST. However, this is generally a very conservative design solution. In Section V, we prove the difficulty of finding an optimal QS for general topologies.

V. Sizing Queues for General Topologies

While fixed QS is a desirable solution, it is unfortunately only optimal for a restricted class of topologies. In this section, we define the QS problem and prove that it is NP-complete by a reduction from vertex cover (VC) [26]. Readers who do not wish to delve into the details of this proof are invited to skip ahead to Section VI, where we discuss relay-station insertion as a solution to throughput degradation.

In the following proof, 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. Thus, a path of length k has k places (but technically, $2 * k$ arcs). Likewise, our figures will now show one arrowhead per edge rather than per arc in contrast to the arcs in Figs. 4–6.

QS Problem:

1) *Instance*: A marked graph G_{QS} modeling a LIS having MST that is equal to $\theta(G_{\mathrm{QS}})$ and an integer K. Let $d[G_{\mathrm{QS}}]$ be the doubled graph of G_{QS}, where every shell has one token per place on its backedges.

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

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.

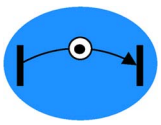


Fig. 7. Vertex construct.

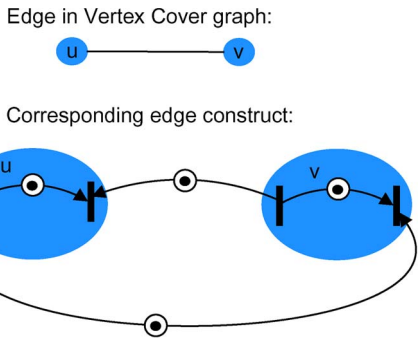


Fig. 8. Edge construct.

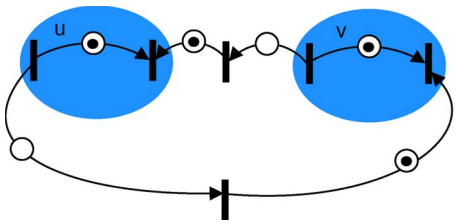


Fig. 9. Edge construct after relay stations have been added.

2) *Question*: Is there a way to add K extra tokens to places on the backedges of $d[G_{\rm QS}]$ such that $\theta(d[G_{\rm QS}]) = \theta(G_{\rm QS})$ (i.e., the MST calculated before adding backedges is the same as the MST after adding backedges)?

Proof That QS Problem is NP-Complete:

1) QS $\in$ NP: Checking a solution to QS can be done with Karp's algorithm to find the minimum cycle mean before and after extra tokens are added to the graph [25], [30]. Karp's algorithm for a graph $G = (V, E)$ has complexity $\bigcirc(|V||E|)$.

2) VC $\propto$ QS: Given an instance of VC, which is a graph $G_{\rm VC} = (V_{\rm VC}, E_{\rm VC})$, and an integer K, we must construct an instance of QS, which is a marked graph $G_{\rm QS}$, and integer K'.

1) First, for every vertex $v \in V_{\rm VC}$, create a *vertex construct* like the one shown in Fig. 7—one edge in $G_{\rm QS}$.
2) Next, for every edge $(u, v) \in E_{\rm VC}$, create an *edge construct* like the one shown in Fig. 8 by adding two edges. All of the transitions in $G_{\rm QS}$ 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 b). Fig. 9 shows the resulting construct for an edge $(u, v) \in E_{\rm VC}$.

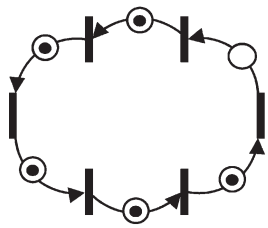


Fig. 10. Cycle that limits the ideal MST to 5/6.

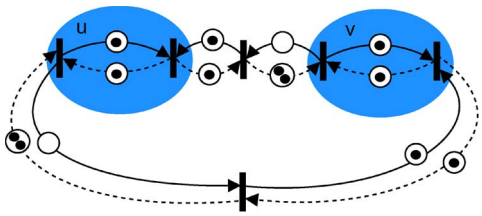


Fig. 11. Edge construct with backedges.

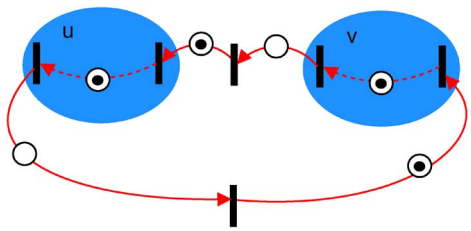


Fig. 12. Cycle in edge construct.

4) Last, add a separate cycle to $G_{\rm QS}$ with six places and five tokens like the one in Fig. 10. This addition sets the MST $\theta(G_{\rm QS})$ to 5/6 since there are no other cycles in the ideal LIS.
5) Let $K' = K$.

To complete the QS problem instance, add in backedges, as shown in Fig. 11. Note that, for every edge $(u, v) \in E_{\rm VC}$, there is a cycle in $G_{\rm QS}$ like the one shown in Fig. 12. This cycle has a mean of $4/6 < 5/6$, causing MST degradation. The only way to avoid this problem is to add exactly one extra token to the backedge of either the u or v vertex construct.

a) Solution to QS $\longrightarrow$ Solution to VC: In this step, we need to show that a solution to the QS instance corresponds to a solution to the 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 $v \in V_{\rm VC}$ has an extra token on its backedge, add v to the cover (i.e., the VC solution).

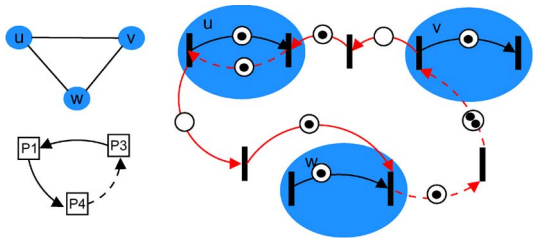


Fig. 13. Example of an additional ("side-effect") cycle.

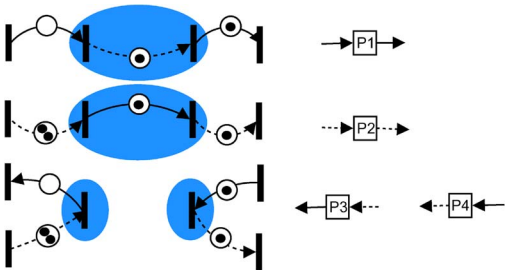


Fig. 14. Four ways to visit (left) a vertex and (right) their P-blocks.

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

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

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

We can separate each of these additional cycles in $G_{\rm QS}$ into two parts: parts that correspond to a vertex construct and parts that do not. There are four ways that a cycle can visit a vertex construct, shown on the left in Fig. 14. Furthermore, because of the way $G_{\rm QS}$ is constructed, between visiting two vertex constructs, the cycle will pass through exactly two places. To help with the clarity of constructing cycles, we represent these different ways of visiting vertex constructs with *P-blocks* (P is for path), shown in Fig. 14. We can build a cycle by connecting P-blocks together. When putting P-blocks together, the matching edges must both be forward or both backward (in the pictures, this means that 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 Fig. 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 $x = \text{tokens}(Px)$, $y = \text{places}(Px)$, $w = \text{tokens}(Py)$, and $z = \text{places}(Py)$, if $x/y \geq 5/6$ and $w/z \geq 5/6$, then $6x \geq 5y$ and $6w \geq 5z$, and therefore, $6(x+w) = 6x + 6w \geq 5y + 6w \geq 5y + 5z = 5(y+z)$. Thus, $(x+w/y+z) \geq 5/6$. Therefore, if we break a cycle up into several paths such that each path has a path mean of at least 5/6, then the cycle mean is at least 5/6. We define *path mean* as the number of tokens in the path divided by the number of places.

TABLE III
TOKENS AND PLACES PER P-BLOCK

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

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 $G_{\rm QS}$ according to the given solution of $G_{\rm VC}$. Because only $P1$ blocks contain backedges from vertex constructs, only $P1$ blocks can ever have extra tokens (while, conveniently, all of the other P-blocks have at least as many tokens as they have places). Now, given an edge, we know that one of its end points must be in the cover. Given a path of k vertices, where k is even, we can break the path up into $k/2$ disjoint edges, and therefore, we can assume that at least $k/2$ 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 $2k/3k$ and then infer $k/2$ extra tokens, and the mean becomes $(2k + (k/2)/3k) = ((4k/2) + (k/2)/3k) = ((5k/2)/3k) = 5k/6k = 5/6$. 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 $(k/2) + 1$ vertices in the VC (integer division); thus, we infer $(k/2) + 1$ extra tokens in the QS graph.

Since only paths with $P1$ blocks can have a path mean of less than 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 earlier, any cycle with an even number of $P1$ blocks has a cycle mean that is equal to 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, which is an even number, of $P1$ blocks, and thus, we know that there are at least $e/2$ extra tokens, bringing the first path's mean up to 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 3/3. Therefore, the overall cycle mean is $\geq 5/6$.

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 number is even, then the path mean is 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

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

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. Thus, 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 six tokens and seven places. Since the cycle can be separated into paths, whose path mean is $\geq 5/6$, the cycle mean must be $\geq 5/6$. Therefore, since we have covered every cycle in the QS instance, a solution to the VC instance corresponds to a solution to the QS instance. This concludes the proof that QS is NP-complete. □

So far, we have discussed QS as a way to reduce MST degradation. An alternative method is to add extra relay stations to the practical LIS. In Section VI, we briefly discuss this technique before returning to QS for algorithms and an empirical evaluation in Sections VII and VIII.

VI. Reducing MST Degradation With Relay-Station Insertion

Additional relay-station insertion to improve system throughput 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" of all reconvergent paths by inserting enough relay stations to make them have the same latency [11]. For instance, adding extra latency to one path of the LIS in Fig. 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. Fig. 15 shows an example. Observe that the system's ideal MST is determined by the cycle $\{A, relay\ station, E, D, C, B, A\}$, whose token-to-place ratio is 5/6. When backedges are considered, the cycle $\{A, relay\ station, E, C, A\}$ reduces the overall system's MST to 3/4. To improve the MST using relay-station insertion, a relay station must be added to either edges (A, C) or (C, E). However, this ends up reducing the system's ideal 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 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 QS. The proof is similar to the proof for QS and is omitted here due to length, but is available in a technical report [21].

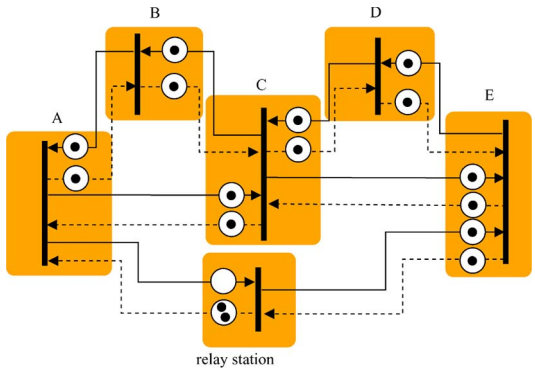


Fig. 15. LIS where relay-station insertion is not enough.

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

VII. Solving the QS Problem

Optimal QS is an NP-complete problem, as proven in Section V. In this section, we discuss how we approach this hard problem by introducing two new algorithms: a heuristic and an exact algorithm. Previous works have used MILP to solve the QS problem [35], [36]. MILP solves the QS problem by framing it as a minimization problem and a series of constraints. Our approach is very different from MILP because we wish to correlate cycles that intersect with each other in the graph, rather than approaching the graph as a whole. We believe that these correlations are helpful because they enable simplification steps that can reduce the problem size by highlighting the differences in coverage that each edge has over cycles in the graph, as we discuss in the next section.

A. Abstraction of QS

We first transform an instance of the QS problem into an instance of the *token deficit (TD) problem*, which is defined formally as follows.

TD Problem:

1) *Instance*. Set of sets $S = (s_1, s_2, s_3, \ldots)$, where each $s_i \in S$ is a set $\{c_i, c_j, \ldots\}$ whose each element has a nonnegative deficit $d(c) \in \mathbf{Z}^*$ and positive integer K.
2) *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 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 the 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, although 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 and all that do not have backedges).
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 the weight of s_i incremented by the deficit of c_x.
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 [35]. 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 TD problem is also NP-complete. This can be shown with a reduction from a dominating set. We do not include the proof but, instead, refer the reader to [20].

B. Algorithms

We propose a heuristic algorithm that produces a solution in $\bigcirc(|S|^2|V||C|)$ 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 TD 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, perform the following:

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.

The cost to check that the weight assignment is correct has complexity $\bigcirc(|S||C|)$, 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 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 $\bigcirc((|S|D)^K)$ 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 the following as inputs: v (number of vertices), s (number of SCCs), c (minimum number of cycles within each SCC), and rs (number of relay stations), whether or not reconvergent paths are allowed between SCCs ($rp = 1$ for yes; zero 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 as follows are the average of 50 trials, where graph topology and the specific locations of relay stations are selected randomly.

A. Relay-Station Insertion and MST Degradation

Backpressure causes degradation of MST 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 have had relay-station insertions, and 3) there are more relay stations than the amount of extra queue space on the backedges. Fig. 16 shows 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

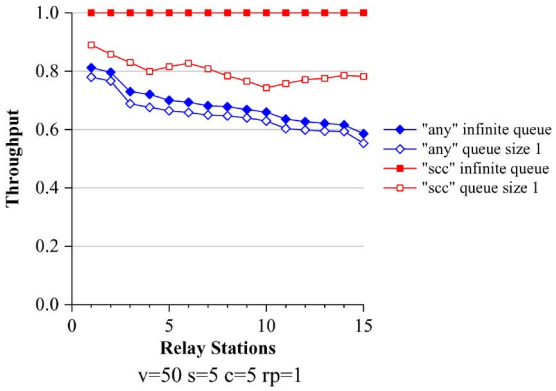


Fig. 16. MST of graph $(v = 50, s = 5, c = 5,$ and $rp = 1)$ given infinite and finite queues.

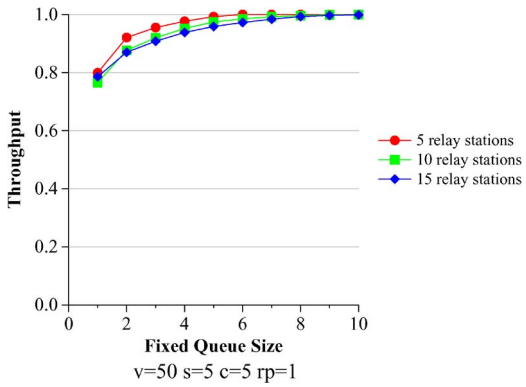


Fig. 17. MST improvement using fixed queues (with scc insertion).

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. 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 remainder of this paper, we focus on graphs that use scc insertion, since this is where the most improvement is needed.

B. Evaluation of Fixed QS

The left system in Fig. 2 is an example of a LIS where optimal MST cannot be maintained with $q = 1$. In fact, while, for a given graph topology and relay-station configuration, it is always possible to size all the queues with a value that is big enough to achieve ideal MST, there is no fixed queue size q that would provide ideal MST for any arbitrary graph topology. To construct a LIS that does not have ideal MST with fixed queues of size q, take Fig. 2 and add $(q - 1)$ more relay stations to the upper channel between A and B.

In extreme cases, fixed QS 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. Fig. 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 75% of the optimal, but when $q >= 5$, it is above 90% of the optimal.

C. Exact Versus Heuristic Solution

Table IV lists the 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. Thus, we can optimize the MST by adding tokens to the inter-SCC edges only. Moreover, since there are no cycles with relay stations and without backedges, we know that the ideal MST is equal to one. Therefore, 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 QS 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 SCCs. "*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.*" is 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*" is the percent of 50 trials that it completed in under an hour. For these cases, "*# Cycles in Unfinished*" and "*Heuristic Soln - no Exact*" are the number of cycles and the heuristic solution, respectively.

The heuristic performs very well in these experiments, producing solutions within 8% of the exact algorithm in every case. 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. One limitation is that the initial listing of all the cycles, which is a necessary step in the heuristic algorithm, may blow up fairly quickly. On an Intel Quad processor with 2 GB of memory, the average time to find all of the cycles when there are fewer than 1000 cycles is 0.22 s (a standard deviation of 0.35); when there are between 1000 and 10 000 cycles, the average time is 2.97 s (a standard deviation of 7.39).

IX. Case Study: A SoC for COFDM Wireless Communication

The heuristic performs well on synthetic systems that are generated using the procedure discussed at the beginning of Section VIII. On the other hand, in this section, we analyze its performance when applied to a real design that is a representative of a class of SoCs for wireless communication applications.

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

(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

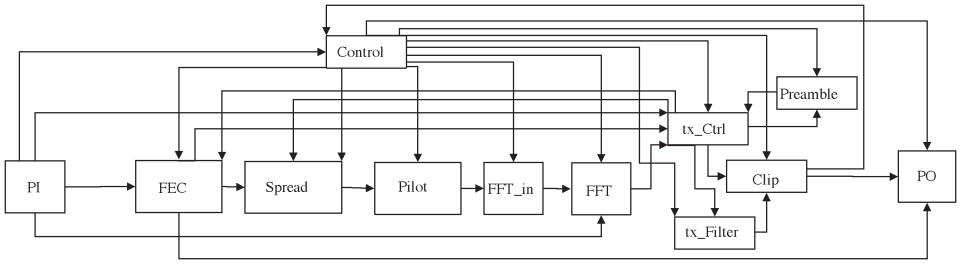


Fig. 18. Case study: An LDPC-COFDM-based UWB transmitter [32], [34].

TABLE V
EXHAUSTIVE INSERTION OF TWO RELAY STATIONS ON EDGES IN THE SYSTEM FROM FIG. 18. WE ASSUME THAT ALL SHELLS HAVE $q = 1$

	Heuristic		Optimal	
Ideal Throughput	0.81			
Actual (Degraded) Thput	0.71			
	Orig.	Simplified	Orig.	Simp.
Solution (extra tokens)	4.00	3.89	3.85	3.84
Average CPU Time (ms)	0.12	0.092	33,000	2.4
Median CPU Time (ms)	0.0050	0.0020	2.4	0.13

Fig. 18 shows the top-level block diagram of a *480-Mb/s coded orthogonal frequency division modulation (COFDM)* transmitter for ultrawideband (UWB) communication [32], [34]. The transmitter receives packets from the medium access control layer, and outputs encoded symbols to a digital–analog converter for physical transmission. Starting from the original RTL specification of the SoC, we derived a corresponding LID version by encapsulating each of the blocks in Fig. 18 with the LID shells presented in [33].

At the top level, the system has 12 blocks, 30 channels, and 22 cycles. After adding backpressure (i.e., doubling the corresponding graph), the number of cycles becomes 2896. The size of the input queues of the LID shell is parameterized. Using Synopsys Design Compiler, we synthesized the original design, as well as the two LID implementations, which have all the shells with uniform queue sizes equal to one and two, respectively. Technology mapping was completed using a 90-nm industrial standard cell library. We performed area estimation and static timing analysis on the mapped netlists. Owing to the relative large size of the various blocks, the critical-path delay of both LID implementations of the transmitter is the same as in the original design, i.e., the maximum clock speed is not affected by the shell encapsulation process. Similarly, the area overhead introduced by the shells with respect to the overall chip design is small both for the case where the shells have queues with uniform size equal to one (1.04% area overhead) and the case where the size is two (3.26% area overhead).

In the design process, locations for relay-station insertion are selected only after floor planning has been carried out. Different optimization criteria can yield different floor plans. Thus, without knowledge of the floor-plan criteria, relay stations could potentially be inserted anywhere in the graph. For the synthetic systems analyzed in Section VIII, we randomly chose locations for relay-station insertion because they present such a large number of edges that an exhaustive search would be infeasible. On the other hand, the top-level diagram of the SoC is small enough for an exhaustive search. There are $\binom{30}{2} = 435$ possible ways to insert two relay stations into the graph (assuming, at most, that one relay station may be placed on each edge). Of these possibilities, 227 (52%) result in throughput degradation.

Table V shows the aggregate statistical results from exhaustively inserting two relay stations on edges in the graph. We ran all tests on an Intel Quad processor with 2 GB of memory. The reported CPU times do not include the time to enumerate cycles. The time to discover all cycles of the graph is 10.5 s. In general, this is an upper bound time, since we must only list cycles with relay stations when analyzing a system with a particular relay-station configuration. The "simplified" system referred to in the table is the system after we have performed the simplification steps described in Section VII-A. The optimal algorithm sometimes returns the solution quickly, but other times, it is very slow. In order to cover all of the cases, we terminated experiments that lasted more than an hour. Two out of 227 (less than 1%) were ended after a timeout both with and without simplification. Of the cases that present throughput degradation and did not time out, the maximum time spent on the optimal algorithm was 57.8 min (10.3 min with simplification), while the minimum was 0.034 ms (0.015 ms). The heuristic algorithm performs very well in these cases, producing a QS solution that, on average, is 4% less efficient than the optimal solution, as listed in Table V. The simplification step can reduce the problem size. In these cases, it reduces the optimal solution on average by a modest 0.01 token. However, after simplification, the heuristic algorithm produces solutions that are only 1.3% less efficient than the optimal solution on average.

To understand better how relay-station insertion can impact throughput in this system, consider the scenario shown in Fig. 19, which corresponds to one of the smaller cases from

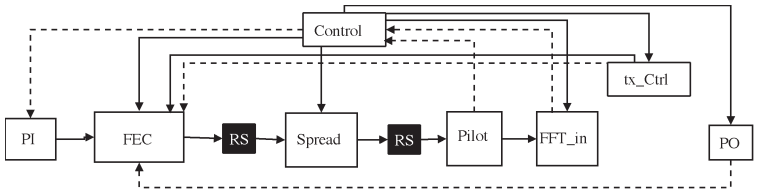


Fig. 19. Subset of the arcs from Fig. 18, which are in cycles that have throughput degradation when relay stations are added between FEC and Spread, and Spread and Pilot, as shown. Backedges are shown as dashed lines.

TABLE VI
POTENTIAL CRITICAL CYCLES FOR THE SCENARIO IN FIG. 19

Cycle	Blocks	Cycle Mean
C_1	(PI, FEC, Spread, Pilot, *Control*, *PI*)	0.71
C_2	(PO, *FEC*, Spread, Pilot, *Control*, PO)	0.71
C_3	(Control, FEC, Spread, Pilot, FFT_in, *Control*)	0.71
C_4	(Control, FEC, Spread, Pilot, *Control*)	0.67
C_5	(Control, tx_Ctrl, FEC, Spread, Pilot, *Control*)	0.71
C_6	(Control, tx_Ctrl, *FEC*, Spread, Pilot, *Control*)	0.71

the exhaustive search. Inserting relay stations on the two edges (FEC, Spread) and (Spread, Pilot) brings down the MST to 0.75 due to the presence of the feedback loop of forward edges: (FEC, Spread, Pilot, FFT_in, FFT, txCtrl, FEC). Moreover, once the graph is doubled, six cycles have a cycle mean lower than 0.75, as reported in Table VI, where the block names that are in italics indicate that the incoming edge to that block is a backedge. Each cycle has a TD equal to one. Hence, adding one additional queue token to one backedge in each cycle is sufficient to increase its cycle mean to at least 0.75. The solution given by both the heuristic and the optimal algorithm is to increase the queue sizes for the backedges (Pilot, Control) and (FFT_in, Control) by one. The extra queue space on edge (Pilot, Control) is sufficient to increase the cycle mean for five of the six cycles listed earlier, while cycle C_3 is taken care by the extra queue space on (FFT_in, Control).

In Section IV, we discussed the possibility of sizing uniformly all the shell queues in the system with the same fixed size q. Table V and Fig. 19 show the throughput degradation for the case when each queue has size q equal to one. When we increase q to be equal to two, none of the cases in our exhaustive search (inserting two relay stations) results in throughput degradation. Thus, even the modest fixed queues can have a beneficial impact. In fact, if only one relay station is inserted into an arbitrary system with $q = 2$, there will never be throughput degradation, because throughput degradation always occurs on cycles that have backedges: If a cycle presents only one relay station and one backedge, having all queues with size equal to two is sufficient to absorb the initialization value τ introduced by the relay station.

X. CONCLUSION

Backpressure is a logical mechanism to control the flow of information on a communication channel and guarantee that no data are lost. Adding backpressure to a LIS, however, can cause a degradation of its 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 yields ideal MST. In more general topologies, using relatively small fixed-size queues can often bring performance within 90% of the ideal MST. However, we also show that the QS problem 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 DAGs of SCCs that only have relay stations between SCCs, can be simplified with a straightforward optimization.

ACKNOWLEDGMENT

The authors would like to thank H.-Y. Liu and C.-Y. Lee for providing the RTL design of the COFDM SoC and C.-H. Li for the help with the LID design of this SoC and the counterexample in Fig. 15.

REFERENCES

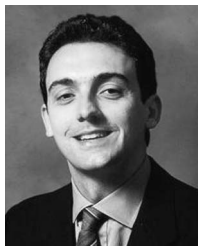
[1] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat, *Synchronization and Linearity*. New York: Wiley, 1992.
[2] A. O. Balkan, M. N. Horak, G. Qu, and U. Vishkin, "Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing," in *Proc. IEEE Symp. High-Perform. Interconnects*, Aug. 2007, pp. 21–28.
[3] A. O. Balkan, G. Qu, and U. Vishkin, "A mesh-of-trees interconnection network for single-chip parallel processing," in *Proc. IEEE 17th Int. Conf. ASAP*, 2006, pp. 73–80.
[4] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous languages twelve years later," *Proc. IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.
[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis From Dataflow Graphs*. Norwell, MA: Kluwer, 1996.
[6] S. M. Burns, "Performance analysis and optimization of asynchronous circuits," Ph.D. dissertation, California Inst. Technol., Pasadena, CA, 1991.
[7] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 1999, pp. 309–315.
[8] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.
[9] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive systems," in *Proc. ACM/IEEE Des. Autom. Conf.*, 2000, pp. 361–367.
[10] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Coping with latency in SOC design," *IEEE Micro*, vol. 22, no. 5, pp. 24–35, Sep./Oct. 2002.
[11] M. R. Casu and L. Macchiarulo, "Issues in implementing latency insensitive protocols," in *Proc. Conf. Des. Autom. Test Eur.*, 2004, pp. 1390–1391.
[12] M. R. Casu and L. Macchiarulo, "A new approach to latency insensitive design," in *Proc. Des. Autom. Conf.*, 2004, pp. 576–581.
[13] M. R. Casu and L. Macchiarulo, "Throughput-driven floorplanning with wire pipelining," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 5, pp. 663–675, May 2005.

[14] V. Chandra, H. Schmit, A. Xu, and L. Pileggi, "A power aware system level interconnect design methodology for latency-insensitive systems," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2004, pp. 275–282.
[15] V. Chandra, A. Xu, H. Schmit, and L. Pileggi, "An interconnect channel design methodology for high performance integrated circuits," in *Proc. Conf. Des. Autom. Test Eur.*, 2004, vol. 2, pp. 1138–1143.
[16] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems," Ph.D. dissertation, Stanford Univ., Palo Alto, CA, 1984.
[17] T. Chelcea and S. M. Nowick, "Robust interfaces for mixed-timing systems with application to latency-insensitive protocols," in *Proc. ACM/IEEE Des. Autom. Conf.*, 2001, pp. 21–26.
[18] P. Cocchini, "Concurrent flip-flop and repeater insertion for high performance integrated circuits," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2002, pp. 268–273.
[19] R. L. Collins and L. P. Carloni, "Topology-based optimization of maximal sustainable throughput in a latency-insensitive system," in *Proc. ACM/IEEE Des. Autom. Conf.*, 2007, pp. 410–415.
[20] R. L. Collins and L. P. Carloni, "Topology-based optimization of maximal sustainable throughput in a latency-insensitive system," Columbia Univ., New York, Tech. Rep. CUCS-008-07, Feb. 2007.
[21] R. L. Collins and L. P. Carloni, "Topology-based performance analysis and optimization of latency-insensitive systems," Columbia Univ., New York, Tech. Rep. CUCS-003-08, Jan. 2008.
[22] F. Commoner, A. W. Holt, S. Even, and A. Pnueli, "Marked directed graphs," *J. Comput. Syst. Sci.*, vol. 5, no. 5, pp. 511–523, Oct. 1971.
[23] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. New York: McGraw-Hill, 2001.
[24] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini, "× *pipes*: A latency insensitive parameterized network-on-chip architecture for multi-processor SoCs," in *Proc. Int. Conf. Comput. Des.*, Oct. 2003, pp. 536–541.
[25] A. Dasdan and R. K. Gupta, "Faster maximum and minimum mean cycle algorithms for system-performance analysis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 10, pp. 889–899, Oct. 1998.
[26] M. R. Garey and D. S. Johnson, *Computers and Intractability*. New York: Freeman, 1979.
[27] M. Geilen, T. Basten, and S. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *Proc. ACM/IEEE Des. Autom. Conf.*, 2005, pp. 819–824.
[28] S. Hassoun and C. J. Alpert, "Optimal path routing in single and multiple clock domain systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 22, no. 11, pp. 1580–1588, Nov. 2003.
[29] J. Hu, Ü. Y. Ogras, and R. Marculescu, "System-level buffer allocation for application-specific networks-on-chip router design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 12, pp. 2919–2933, Dec. 2006.
[30] R. Karp, "A characterization of the minimum cycle mean in a digraph," *Discrete Math.*, vol. 23, no. 3, pp. 309–311, Sep. 1978.
[31] S. Kim and P. A. Beerel, "Pipeline optimization for asynchronous circuits: Complexity analysis and an efficient optimal algorithm," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2000, pp. 296–302.
[32] C.-Y. Lee, H.-Y. Liu, and C.-C. Lin, "SoC for COFDM wireless communications: Challenges and opportunities," in *Proc. Int. Symp. VLSI Des. Autom. Test*, 2006, pp. 1–4.
[33] C.-H. Li, R. L. Collins, S. Sonalkar, and L. P. Carloni, "Design, implementation, and validation of a new class of interface circuits for latency-insensitive design," in *Proc. Int. Conf. Formal Methods Models Codesign*, 2007, pp. 13–22.
[34] H.-Y. Liu, C.-C. Lin, Y.-W. Lin, C.-C. Chung, K.-L. Lin, W.-C. Chang, L.-H. Chen, H.-C. Chang, and C.-Y. Lee, "A 480 mb/s LDPC-COFDM-based UWB baseband transceiver," in *Proc. ISSCC Tech. Dig. Papers*, 2005, vol. 1, pp. 444–609.
[35] R. Lu and C.-K. Koh, "Performance optimization of latency insensitive systems through buffer queue sizing of communication channels," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2003, pp. 227–231.
[36] R. Lu and C.-K. Koh, "Performance analysis of latency-insensitive systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 3, pp. 469–483, Mar. 2006.
[37] R. Lu, G. Zhong, C.-K. Koh, and J.-Y. Chao, "Flip-flop and repeater insertion for early interconnect planning," in *Proc. Conf. Des. Autom. Test Eur.*, Mar. 2002, pp. 690–695.
[38] R. Manohar and A. J. Martin, "Slack elasticity in concurrent computing," in *Proc. Math. Program Constr.*, 1998, pp. 272–285.
[39] A. Maxiaguine, S. Künzli, S. Chakraborty, and L. Thiele, "Rate analysis for streaming applications with on-chip buffer constraints," in *Proc. Asia S. Pacific Des. Autom. Conf.*, 2004, pp. 131–136.
[40] T. Murata, "Circuit theoretic analysis and synthesis of marked graphs," *IEEE Trans. Circuits Syst.*, vol. CAS-24, no. 7, pp. 400–405, Jul. 1977.
[41] T. Murata, "Petri Nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.
[42] P. Pan, A. Karandikar, and C. L. Liu, "Optimal clock period clustering for sequential circuits with retiming," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 6, pp. 489–498, Jun. 1998.
[43] P. Poplavko, T. Basten, M. Bekooij, J. L. van Meerbergen, and B. Mesman, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," in *Proc. Int. Conf. Compilers, Architectures Synthesis Embedded Syst.*, 2003, pp. 63–72.
[44] P. Prakash and A. J. Martin, "Slack matching quasi delay-insensitive circuits," in *Proc. Int. Symp. Asynchr. Syst. Circuits*, Mar. 2006, pp. 30–39.
[45] C. V. Ramamoorthy and G. S. Ho, "Performance evaluation of asynchronous concurrent systems using Petri nets," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 5, pp. 440–449, Sep. 1980.
[46] R. Reiter, "Scheduling parallel computations," *J. ACM*, vol. 15, no. 4, pp. 309–311, Oct. 1968.
[47] L. Scheffer, "Methodologies and tools for pipelined on-chip interconnect," in *Proc. Int. Conf. Comput. Des.*, Oct. 2002, pp. 152–157.
[48] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Proc. Des. Autom. Conf.*, 2007, pp. 777–782.
[49] G. Venkataramani and S. C. Goldstein, "Leveraging protocol knowledge in slack matching," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2006, pp. 724–729.
[50] T. Yamada and S. Kataoka, "On some LP problems for performance evaluation of timed marked graphs," *IEEE Trans. Autom. Control*, vol. 39, no. 3, pp. 696–698, Mar. 1994.

Rebecca L. Collins received the B.S. (*summa cum laude*) and M.S. degrees in computer science from the University of Tennessee, Knoxville, in 2003 and 2004, respectively. She is currently working toward the Ph.D. degree in computer science at the Columbia University, New York, NY.

Her research interests include performance optimization and load balancing in multicore systems, queue sizing, and other performance enhancements of latency-insensitive systems.

Ms. Collins was a recipient of the National Defense Science and Engineering Graduate fellowship.



Luca P. Carloni (S'95–M'04) received the Laurea degree (*summa cum laude*) in electrical engineering from Università di Bologna, Bologna, Italy, in 1995, and the M.S. and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 1997 and 2004, respectively.

He is currently an Assistant Professor with the Department of Computer Science, Columbia University, New York, NY. He has authored over 50 publications and is the holder of one patent. His research interests are in the area of design tools and methodologies for integrated circuits and systems, distributed embedded system design, and design of high-performance computer systems.

Dr. Carloni is a member of the IEEE Computer Society. He received the Faculty Early Career Development Award from the National Science Foundation in 2006 and was selected as an Alfred P. Sloan Research Fellow in 2008. He is the recipient of the 2002 Demetri Angelakos Memorial Achievement Award "in recognition of altruistic attitude toward fellow graduate students." In 2002, one of his papers was selected for "The Best of International Conference on Computer-Aided Design (ICCAD)," which is a collection of the best IEEE ICCAD papers of the past 20 years.