Topology-Based Optimization of Maximal Sustainable Throughput in a Latency-Insensitive System

Rebecca L. Collins
Dept. of Computer Science
Columbia University
rlc2119@cs.columbia.edu

Luca P. Carloni
Dept. of Computer Science
Columbia University
luca@cs.columbia.edu

ABSTRACT

We consider the problem of optimizing the performance of a latency-insensitive system (LIS) where the addition of backpressure has caused throughput degradation. Previous works have addressed the problem of LIS performance in different ways. In particular, the insertion of relay stations and the sizing of the input queues in the shells are the two main optimization techniques that have been proposed. We provide a unifying framework for this problem by outlining which approaches work for different system topologies, and highlighting counterexamples where some solutions do not work. We also observe that in the most difficult class of topologies, instances with the greatest throughput degradation are typically very amenable to simplifications. The contributions of this paper include a characterization of topologies that maintain optimal throughput with fixed-size queues and a heuristic for sizing queues that produces solutions close to optimal in a fraction of the time.

Categories and Subject Descriptors

B.5.2 [**Register-transfer-level implementation**]: [Design Aids.]; F.1.2 [**Computation by Abstract Devices**]: [Parallelism and Concurrency.]

Keywords

Latency-Insensitive Design, Performance Analysis.

General Terms

Algorithms, Performance.

1. INTRODUCTION

Latency-insensitive design (LID) is a design methodology for system-on-chip (SOC) that simplifies the assembly of IP cores and enables automatic wire pipelining up to late stages of the design process [2]. Given a netlist of IP cores specified in synthesizable RTL format, a latency-insensitive system (LIS) can be automatically derived by encapsulating each core within a *shell*, an automatically-synthesized block that acts as an interface for global, i.e. inter-core, communication. The idea is to build a distributed communication architecture 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) 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 without requiring the redesign of any IP core and without jeopardizing the system behavior [3]. Essentially, this is possible because: (1) the data exchanged by the shells are marked as either valid or void, (2) the relay stations are initialized with void data, (3) each shell keeps its core "unaware" of the existence of void data by controlling it through an *AND-firing* policy: at each clock cycle the 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 the same exact sequence of valid data modulo the void data [3].

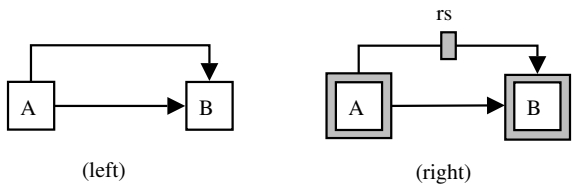

Figure 1: A simple SOC transformed into an LIS. A and B are encapsulated in shells, and a relay station is inserted on A's upper channel.

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 1: Output of LIS components in Fig. 1(right).

Figure 1 shows a simple SOC and how it is transformed into an LIS: A and B are IP Cores encapsulated in LIS shells. We assume that the upper channel has been routed on a path much longer than the lower channel and that this has forced us to wire pipeline it through the insertion of a relay station rs in order to meet the target clock period. The output behavior of the three components is shown in Table 1: A 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 0. We use τ to denote a void data token as proposed in [3].

The main performance metric of an LIS is the rate of production of valid data. This data throughput depends on two factors: the internal structure of the system and the interaction with the environment where it operates. The inter-

DAC 2007, June 4–8, 2007, San Diego, California, USA.

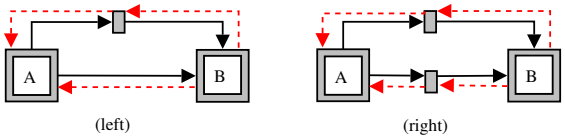

Figure 2: Backedges in an LIS (left). Adding a relay station for performance (right).

nal structure determines its *maximal sustainable throughput (MST)* θ as the system effectively processes 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 the system reduces θ because its initialization τ continues to circulate around the loop and causes each shell on the loop to periodically stall its core [4, 12, 13]. In the example of Fig. 1 there is no feedback loop and the τ produced by the relay station in the first timestep passes out of the system, which therefore has $\theta = 1$. Note, however, that the shell of B must still buffer the first valid value from A's lower channel, 1, in the corresponding input queue while the first data value from A's upper channel, 0, spends a cycle in the relay station.

If this simple system does not interact with the environment, a queue of size one is sufficient to avoid any data loss. In general, however, systems are composed to derive more complex systems: this makes it impossible to know in advance the sequence of τ tokens that each component will observe. For instance, if one uplink subsystem with an MST of $\frac{3}{4}$ feeds another downlink subsystem with an MST of $\frac{2}{3}$, only a queue of infinite size could avoid loss of data. But since infinite queues are not realizable in practice, a shell like B may not have enough room to buffer all of the incoming data. Hence, it must be able to send a *stop* signal back on any input channel to indicate that its queue is full and that the uplink shell on the channel must stall. This operation, called *backpressure* [3], causes an "implicit" feedback loop and the system will have $\theta < 1$. We illustrate backpressure feedback loops in Fig. 2 by adding a backward edge (*backedge*) for every forward edge of our example. 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 B has space to store one data token from A, it still must stall A after filling the space since B does not know beforehand when the relay station sends valid data: if the relay station sends a τ when B's lower input channel queue is already full, B may miss a valid data token from A.

MST calculation is described in Section 2. We call a graph G with backedges, the "double" of G, $d[G]$. The MST of G is denoted $\theta(G)$. It has been shown that $\theta(G) = \theta(d[G])$ when the system theoretically has infinite queues and practically *big enough* finite queues [13]. However, it is a challenge to determine how big finite queues must be to match the performance of a system with infinite queues.

Sometimes an alternative to increasing queue size is to insert additional relay stations that would not be required for wire pipelining but that are useful to increase the MST. For instance, we can insert a relay station on the second channel as shown in Fig. 2 (right). Now A's data is delayed one cycle along both channels, and B receives data from both channels at the same time. This approach allows more flexible placement of the additional storage space, but the additional relay stations can potentially impact performance elsewhere in the system. In Section 3.2, we present a system whose MST cannot be optimized by only adding relay stations.

Contributions. We focus on the performance optimization of a *practical* LIS (with backpressure and finite queues) so that its MST is equal to the MST of an equivalent *ideal* LIS (with infinite queues and no backpressure). In Sec. 3.1, we show that setting every queue size to one is sufficient for

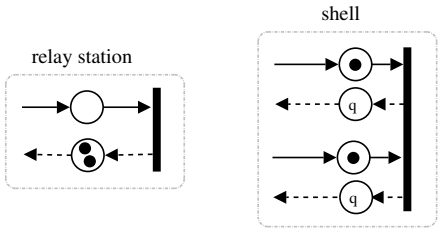

Figure 3: Modeling relay station and shell with marked graphs.

graphs that do not have reconvergent paths. Sec. 3.2 explains the limitations of the method based on relay station insertion. When fixed queues and relay station insertion cannot improve the MST, the queues must be increased. In Sec. 4 we discuss the complexity of the queue sizing problem and we present a heuristic algorithm that produces solutions close to the optimal solutions in a fraction of the time. In Sec. 5 we evaluate it with experimental results.

Related Work. Carloni *et al.* have proposed latency-insensitive design [2, 3] and analyzed the throughput performance assuming that the system has infinite queues [4]. Lu and Koh showed that the performance of an LIS with finite queues can match the performance of an LIS with infinite queues if the queues are big enough [12, 13]. They presented a mixed integer linear programming solution to queue sizing. We share the same scope, but focus instead on topology and the simplifications and assumptions that can be made when something is known about the structure of a system. Casu and Macchiarulo avoided queue sizing issues by scheduling the activation of blocks and eliminating backpressure [6, 7]. A limitation of their work is that building schedules requires that each block has knowledge about the global system behavior.

2. LIS AS A MARKED GRAPH

The components of a latency-insensitive system produce data or τ tokens synchronously according to a global clock. Marked graphs are a natural model for data movement, and we use them to evaluate throughput performance of LISs.

A marked graph is a bipartite directed graph with two kinds of vertices: *places*, and *transitions*. By definition, each place has exactly one incoming edge and one outgoing edge that both go to transitions. Places have the ability to hold 0 or more *tokens*. Transitions cannot hold tokens, but they can *fire* and move tokens around in the graph. Each incoming edge to a transition comes from a place, and each outgoing edge goes to a place. 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 [9].

While the overall number of tokens in a marked graph can change, the number of tokens in the edges of a cycle never changes [14]. For a complete description of marked graph properties, the reader can consult [14, 15]. For our purposes we slightly restrict marked graphs from their original form and we assume that they are both synchronous and deterministic. The deterministic assumption forces each enabled transition to fire (in the original definition, an enabled transition may fire nondeterministically and independently of other transitions). The synchronous assumption forces the firing of all "simultaneously-enabled" transitions to occur concurrently according to a "global clock".

Figure 3 shows the marked graph representation of a relay station and a shell with backpressure where $q = 1$ (ie. shell queues are size 1). The large white circles represent places, the small black circles represent tokens, and the rectangles

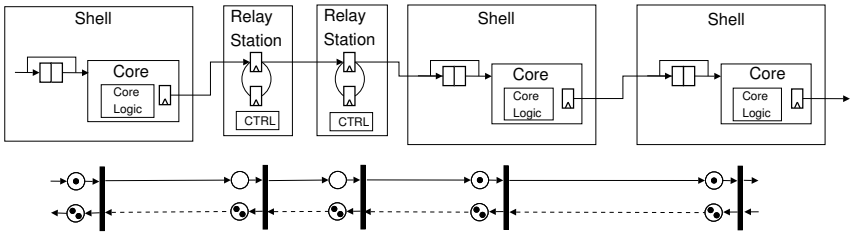

Figure 4: Marked-graph model (with $q = 2$) of a path across multiple shells and relay stations in a LIS.

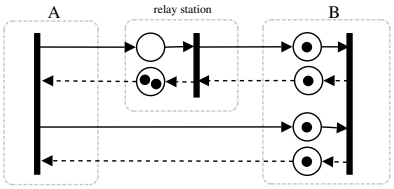

Figure 5: Marked-graph model of the LIS of Fig. 1, with $q = 1$.

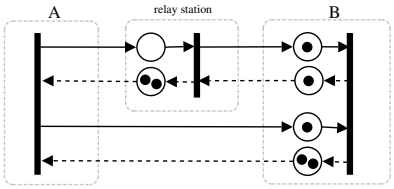

Figure 6: Queue sizing solution to the throughput degradation shown in Figure 5.

represent transitions. Initially, the relay station's incoming forward edge has no token since the relay station must produce a τ in the first timestep, and its outgoing backedge has 2 tokens corresponding to the 2 available slots in the queue. The shell's incoming forward edges each have one token since the shell must produce a valid data token in the first timestep, and its backedges have q tokens (shown with the symbol q). Using a marked graph representation, informative data tokens are represented by marked graph tokens on the forward edges. The tokens on the backedges (shown as dashed lines in the figure) instead represent available space in the 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 similarly 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. Also, a shell relies on the stallable core logic to latch the output signals and features by-passable input queues to avoid adding any cycle 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 shell-core pair or a relay station is one clock period. In the marked-graph model the various data storage elements in each module are abstracted and a place 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.

The *initial marking* of a graph, G, specifies how many tokens each place has at initialization. Based on this, we can compute the MST $\theta(G)$ of the graph by finding the cycle with the lowest ratio of tokens to places [1]. Figure 5 shows the marked graph representation of the LIS in Figure 1 assuming $q = 1$. The cycle $\{A, relay\ station, B, A\}$ has 3 places but only 2 tokens, and so the system has MST of $\frac{2}{3}$. To restate the problem of queue sizing, adding backpressure may create new cycles that have lower token-to-place ratio. But the number of tokens in backedges can be altered by increasing the queues, so if enough tokens are added to the doubled graph, the MST of the graph will match the lowest ratio $t : p$ in the undoubled graph. In Figure 6, the queue for B's lower channel is increased to 2, and now the system has optimal MST. We call this problem the *Token Deficit Problem (TDP)*, since we must decide which places in the marked graph should have more tokens. Section 4 formalizes the TDP and presents a heuristic for solving it.

3. WILL FIXED QUEUE SIZING WORK?

Fixed queue sizing is setting all queues in a system to the same length. In the example illustrated in Figure 5, the queue sizes are fixed with $q = 1$. 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. The length $|p|$ of a path 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 simple paths is *reconvergent* if they would form a cycle if the graph were undirected. The *strongly connected components* (SCCs) of a directed graph are partitions of the vertices such that all sets of vertices in an SCC are mutually reachable. An *articulation point* is a vertex without which the graph will be disconnected [10].

3.1 SCC and No Reconvergent Paths

Claim: A practical LIS 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. Since G is strongly connected, there is a path from u to v and a path from v to u. If the path from u to v is p_1 and the path from v to u is p_2, there can't 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. 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 G's original cycles, where the inverse of cycle c is the cycle formed by the backedges of all of c's edges. All backedges have at least one token. Thus, we are guaranteed that the inverse of cycle c has at least as many tokens as c has, and 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, an 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 with no reconvergent paths.

3.2 Limitations of Relay Station Insertion

Relay stations can be added to LISs 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 [5].

However, there are some systems for which no assignment of additional relay stations can optimize performance. Figure 7 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 backpressure edges 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 $\frac{3}{4}$.

In the cases where relay station insertion is not enough to solve our throughput problems, and where we cannot avoid reconvergent paths, we turn to queue sizing techniques.

4. SIZING QUEUES

In this section, we describe our algorithms for sizing queues in an LIS. We approach the problem of sizing queues by abstracting away the edges. In this form, the problem becomes NP-complete. Previous approaches [12] solved the problem with mixed integer linear programming (also NP-hard). In general, no easy solution is known for the problem of queue sizing.

4.1 Token Deficit Problem (TDP)

We call our abstraction of queue sizing the *Token Deficit Problem*. The TDP is the problem of filling the token deficits of cycles in an LIS graph. Let the cycles be partitioned 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. We formalize the TDP below and sketch its proof of NP-completeness. Creating an instance of TDP from an instance of queue sizing requires a listing of the cycles of the graph. This step is potentially exponential, though in many practical cases the number of cycles is not large. We mitigate these costs by simplifying the LIS graphs where possible: e.g., if a graph 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 solve the simplified graph - greatly reducing the number of cycles that must be enumerated. This particular case is discussed in more detail in Section 5.1, and we show in Section 5.3 that our heuristic solution performs well for larger graphs of this type.

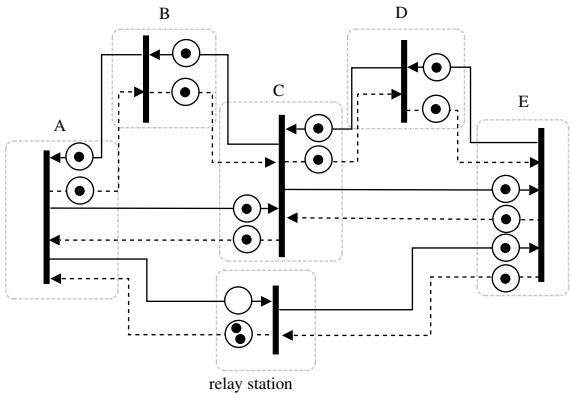

Figure 7: An LIS where relay-station insertion is not enough.

Token Deficit Problem:
Instance: Set $S = (s_1, s_2, s_3, ..)$ 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 $c_i \in s_i$?

There always exists a K for which the Token Deficit Problem can be solved [12]. An easy way to look at this is to consider that every relay station can insert at most one τ, or bubble into the system, and if there are R relay stations, no cycle can be deficient more than R tokens. Adding R extra tokens to one edge in every cycle of the graph that has backedges guarantees that no cycle with backedges will have a cycle mean less than 1.

The Dominating Set Problem [11] is the problem of choosing vertices in a graph such that every vertex is either in the Dominating Set or a neighbor of a vertex in the Dominating Set. Dominating Set is NP-complete and can be reduced to Token Deficit with the following construction: Given an instance of the Dominating Set problem, i.e. a graph $G = (V, E)$, let S be a set of size $|V|$, such that each element s_i corresponds to a vertex $v_i \in V$, and $s_i = v_i \cup \{$all vertices v_j that are adjacent to v_i in $G\}$. Assign each v_i a deficit $d(v_i) = 1$. For each s_i, its weight assignment $w(s_i)$ is equal to 1 if $v_i \in$ *Dominating Set* and to 0 otherwise. The proof that there is Dominating Set of size K if and only if there is a Token Deficit Solution whose overall weight assignments are K is given in [8].

4.2 Solving Token Deficit

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. We evaluate the heuristic in Section 5.

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.

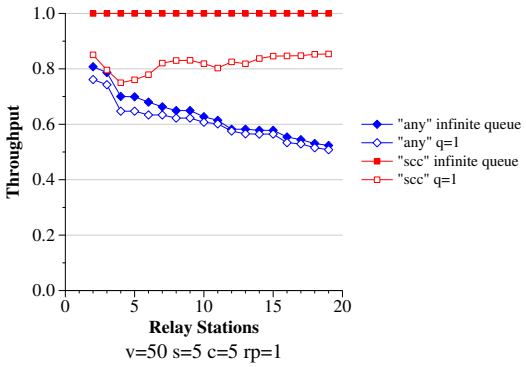

Figure 8: MST of graph (v=50,s=5,c=5,rp=1) given infinite and finite ($q = 1$) queues.

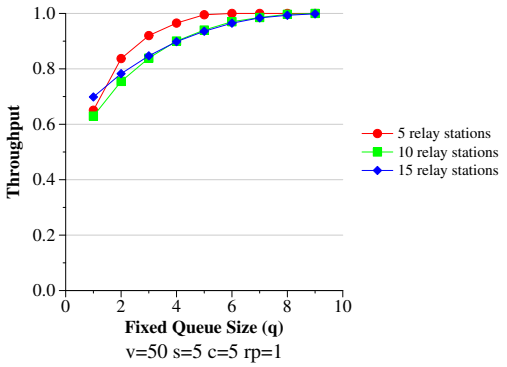

Figure 9: MST Improvement using Fixed Queues.

2. If any $w(s_i)$ remains unfixed, repeat Step 1. 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. For comparison purposes, we also develop an algorithm that produces the optimal solutions to TDP. 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 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.

Preprocessing Optimizations. Before running the heuristic, the following preprocessing optimizations can greatly reduce the problem size. First, if set s' is a subset of another set s, delete set s'. Recall that the set s corresponds to an edge, e_s in the LIS, and the elements of s are the cycles that go through that edge. If s' is a subset of s, then the edge e_s is already on all of the cycles that $e_{s'}$ is on. We can assume for simplicity that the extra queue space is all added to the e_s, and ignore $e_{s'}$. Next, if a cycle c only occurs in a set s, the queue of the edge that corresponds to s can be incremented by a quantity equal to the deficit of c.

5. EXPERIMENTS

To evaluate our heuristic algorithm we made 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 (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 v nodes into *scc* partitions;
2. for each strongly-connected component, s'
   (a) make a cycle that visits all of the vertices in s';
   (b) choose $u, v \in s'$ s.t. (u, v) has not already been added to the graph, and add (u, v) to the graph;
   (c) repeat step 2b (*cycles* - 1) times; this guarantees that at least *cycles* 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 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 the connected temporary graph, 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.

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

5.1 MST Degradation

Backpressure causes a degradation of maximal sustainable throughput in cases where a graph contains a cycle that is made up of both backedges and forward edges and one or more of the forward edges in the cycle has had relay station insertions, *and* where there are more relay stations than the amount of extra queue space on the backedges. Figure 8 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. 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 possible.

5.2 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 an 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 queues can be a fast and

(V,E)	# SCC	# Edges (inter-SCC)	Cycles (inter-SCC)	RS	Exact Soln.	Heuristic Soln.	% Exact finished	Unfinished cycles	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 2: How good are the heuristic solutions?

effective approach. Figure 9 shows the MST improvements that are gained in LIS derived with our graph generator as the fixed queue size q increases. On average, with $q = 1$, MST can be as low as 65% of the optimal, but when $q >= 5$, MST is above 90% of the optimal.

5.3 Exact vs. Heuristic Solution

Table 2 lists the results of several experiments using LIS 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 since 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 MST by adding tokens to the inter-SCC edges only. Second, since there are no cycles with relay stations 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 number of edges between SCC. *Exact Soln.* lists the average number of additional queue space (tokens added to the marked graph representation) that are necessary to optimize performance using the exact algorithm. *Heuristic Soln.* shows the average number of queue space needed 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 *unfinished cycles* 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 solution, 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.

6. CONCLUDING REMARKS

Adding backpressure to a latency-insensitive system 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 study how the topology of a system can impact the MST degradation, and how it is related to the different solutions. When a system is made up of SCCs with no reconvergent paths, or a tree of SCCs with no reconvergent paths, using fixed queues with size $q = 1$ achieves optimal MST. In more general topologies, using relatively small fixed size queues can often bring performance within 90% of the optimal. We present a heuristic that is guaranteed to produce a performance-wise optimal solution that may require slightly more queue space than the exact solution. Interestingly enough, we show that the class of graphs with the worst 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.

Acknowledgments

This work has been partially sponsored by an NDSEG fellowship and the GSRC.

7. REFERENCES

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

[2] 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.

[3] 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.

[4] 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. ACM, 2000.

[5] 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.

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

[7] 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.

[8] 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, February 2007.

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

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

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman & Co., New York, 1979.

[12] R. Lu and C. 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, Washington, DC, USA, 2003. IEEE Computer Society.

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

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

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