Report Number: WUCSE-2004-6

2004-02-07

Work-Conserving Distributed Schedulers

Prashanth Pappu, Jonathan S. Turner, and Ken Wong

Buffered multistage interconnection networks offer one of the most scalable and cost-effective approaches to building high capacity routers and switches. Unfortunately, the performance of such systems has been difficult to predict in the presence of the extreme traffic conditions that can arise in Internet routers. Recent work introduced the idea of distributed scheduling, to regulate the flow of traffic in such systems. This work demonstrated (using simulation and experimental measurements) that distributed scheduling can en-able robust performance, even in the presence of adversarial traffic patterns. In this paper, we show that appropriately designed distributed scheduling algorithms are provably work-conserving... Read complete abstract on page 2.

Work-Conserving Distributed Schedulers

Prashanth Pappu, Jonathan S. Turner, and Ken Wong

Complete Abstract:

Buffered multistage interconnection networks offer one of the most scalable and cost-effective approaches to building high capacity routers and switches. Unfortunately, the performance of such systems has been difficult to predict in the presence of the extreme traffic conditions that can arise in Internet routers. Recent work introduced the idea of distributed scheduling, to regulate the flow of traffic in such systems. This work demonstrated (using simulation and experimental measurements) that distributed scheduling can en-able robust performance, even in the presence of adversarial traffic patterns. In this paper, we show that appropriately designed distributed scheduling algorithms are provably work-conserving for speedups of 2 or more. Two of the three algorithms presented were inspired by algorithms previously developed for crossbar scheduling. The third has no direct counterpart in the crossbar scheduling context. In our analysis, we show that distributed schedulers based on blocking flows in small-depth acyclic flow graphs can be work-conserving, just as certain crossbar schedulers based on maximal bipartite matchings have been shown to be work-conserving. We also study the performance of practical variants of the work-conserving algorithms with speedups less than 2, using simulation. These studies demonstrate that distributed scheduling ensures excellent performance under extreme traffic conditions for speedups of less than 1.5.

2004-6

Work-Conserving Distributed Schedulers

Authors: Pappu, Prashanth; Turner, Jonathan; Wong, Ken

February 7, 2004

Abstract: Buffered multistage interconnection networks offer one of the most scalable and cost-effective approaches to building high capacity routers and switches. Unfortunately, the performance of such systems has been difficult to predict in the presence of the extreme traffic conditions that can arise in Internet routers. Recent work introduced the idea of distributed scheduling, to regulate the flow of traffic in such systems. This work demonstrated (using simulation and experimental measurements) that distributed scheduling can en-able robust performance, even in the presence of adversarial traffic patterns. In this paper, we show that appropriately designed distributed scheduling algorithms are provably work-conserving for speedups of 2 or more. Two of the three algorithms presented were inspired by algorithms previously developed for crossbar scheduling. The third has no direct counterpart in the crossbar scheduling context. In our analysis, we show that distributed schedulers based on blocking flows in small-depth acyclic flow graphs can be work-conserving, just as certain crossbar schedulers based on maximal bipartite matchings have been shown to be work-conserving. We also study the performance of practical variants of the work-conserving algorithms with speedups less than 2, using simulation. These studies demonstrate that distributed scheduling ensures excellent performance under extreme traffic conditions for speedups of less than 1.5.

Work-Conserving Distributed Schedulers for Terabit Routers

Prashanth Pappu, Jonathan Turner and Ken Wong
{prashant,jst,kenw}@cse.wustl.edu
WUCSE-04-06

February 7, 2004

Department of Computer Science and Engineering
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

Buffered multistage interconnection networks offer one of the most scalable and cost-effective approaches to building high capacity routers and switches. Unfortunately, the performance of such systems has been difficult to predict in the presence of the extreme traffic conditions that can arise in Internet routers. Recent work introduced the idea of distributed scheduling, to regulate the flow of traffic in such systems. This work demonstrated (using simulation and experimental measurements) that distributed scheduling can enable robust performance, even in the presence of adversarial traffic patterns. In this paper, we show that appropriately designed distributed scheduling algorithms are provably *work-conserving* for speedups of 2 or more. Two of the three algorithms presented were inspired by algorithms previously developed for crossbar scheduling. The third has no direct counterpart in the crossbar scheduling context. In our analysis, we show that distributed schedulers based on *blocking flows* in small-depth acyclic flow graphs can be work-conserving, just as certain crossbar schedulers based on maximal bipartite matchings have been shown to be work-conserving. We also study the performance of practical variants of the work-conserving algorithms with speedups less than 2, using simulation. These studies demonstrate that distributed scheduling ensures excellent performance under extreme traffic conditions for speedups of less than 1.5.

This work was supported by the Defense Advanced Research Projects Agency (contract N660001-01-1-8930).

Work-Conserving Distributed Schedulers for Terabit Routers

Prashanth Pappu, Jonathan Turner and Ken Wong
{prashant,jst,kenw}@cse.wustl.edu

1. Introduction

High performance routers must be scalable to hundreds or even thousands of ports. The most scalable architectures for high capacity routers include systems using multistage interconnection networks with internal buffers and a small speedup relative to the external links; that is, the internal data paths operate at speeds that are faster than the external links by a small constant factor (typically between 1 and 2). In the presence of a sustained overload at an output port, such systems can become congested with traffic attempting to reach the overloaded output, interfering with the flow of traffic to other outputs. The unregulated nature of traffic in IP networks makes such overloads a normal fact of life, which router designers must address, if their systems are to be robust enough to perform well under the most demanding traffic conditions.

Reference [11] introduced the use of distributed scheduling to manage the flow of traffic through a large router in order to mitigate the worst effects of the most extreme traffic conditions. Distributed scheduling borrows ideas developed for scheduling packet transmissions through crossbar switches [2,5,7,8]. The core idea is to use *Virtual Output Queues* (VOQ) at each input. That is, each input maintains separate queues for each output. (Queues are implemented as linked lists, so the only per queue overhead is for the queues' head and tail pointers.) Packets arriving at inputs are placed in queues corresponding to their outgoing links. In crossbar scheduling, a centralized scheduler selects packets for transmission through the crossbar, seeking to emulate, as closely as possible, the queueing behavior of an ideal output queued switch. The centralized scheduler used in crossbar scheduling makes scheduling decisions every packet transmission interval. For routers with 10 Gb/s links, this typically means making scheduling decisions every 40 ns, a demanding requirement, even for a router with a small number of links. For larger routers it makes centralized scheduling infeasible.

Distributed scheduling, unlike crossbar scheduling, does not seek to schedule the transmission of individual packets. Instead, it regulates the number of packets forwarded during a period which we call the *scheduling interval* and denote by *T*. The scheduling interval is typically fairly long, on the order of tens of microseconds. The use of such coarse-grained scheduling means that a distributed scheduler can only approximate the queueing behavior of an ideal output-queued switch, but does allow systems to scale up to larger configurations than are practical with fine-grained scheduling. In a router that implements distributed scheduling, the *Port Processors* (the components that terminate the external links, make routing decisions and queue packets) periodically exchange information about the status of their VOQs. This information is then used to rate control the VOQs, with the objective of moving packets to the output side of the router as expeditiously as possible, while avoiding congestion within the interconnection network. So long as the scheduling interval is kept small relative to end-to-end delays (which are typically tens to hundreds of milliseconds in wide area networks) the impact of coarse scheduling on the delays experienced by packets can be acceptably small.

While [11] demonstrated, using simulation and experimental measurement, that distributed scheduling could ensure excellent performance under extreme traffic conditions, it could provide no analytical bounds on the performance of the proposed algorithms, nor a rigorous justification for the specific design choices that were made. This paper corrects that deficiency, by showing that there are distributed scheduling algorithms that are provably *work-conserving*, for speedups of 2 or more. The analysis provides insight that motivates the design of more practical variants of these algorithms, which provide excellent performance, significantly improving upon the performance reported in [11]. Where the algorithms described in [11] can fail to be work-conserving, with speedups of more than 2, the algorithms reported here are demonstrably work-conserving for extreme traffic, even when speedups are limited to 1.5. One interesting aspect of the analysis is the role played by network flows, which parallels the role played by bipartite matching in crossbar scheduling. Specifically, we show that distributed schedulers based on finding *blocking flows* in small depth acyclic flow graphs and that favor outputs with short queues are work-conserving, much as crossbar schedulers based on finding maximal matchings in bipartite graphs that favor outputs with short queues are work-conserving.

While distributed scheduling shares some features of crossbar scheduling, it differs in two important respects. First, the distributed nature of these methods rules out the use of the iterative matching methods that have proved effective in crossbar scheduling, since each iteration would require an exchange of information, causing the overhead of the algorithm to increase in proportion to the number of iterations. On the other hand, the shift to coarse scheduling provides some flexibility that is not present in crossbar scheduling. In crossbar scheduling, it is necessary to match inputs and outputs in a one-to-one fashion during each scheduling operation. In distributed scheduling, we allocate the interface bandwidth at each input and output and are free to subdivide that bandwidth in whatever proportions will produce the best result.

Recently, there has been considerable interest in a switch architecture called the *load balanced switch* described in [4] and used in [6]. This architecture consists of a single stage of buffers sandwiched between two identical stages of switching, each of which walks through a fixed sequence of configurations. The fixed sequence of switch configurations makes the switching components very simple and the system is capable of achieving 100% throughput for random traffic. Unfortunately, this architecture also has a significant drawback. To avoid resequencing errors, each output requires a resequencing buffer capable of holding about n^2 packets. These buffers impose a delay that grows as the square of the switch size. For the 600 port switch described in [6], operated with a switching period of 100 ns, this translates to a delay of about 36 milliseconds, a penalty which applies to all packets, not just to an occasional packet.

The paper is organized as follows. Section II introduces two scheduling methods, proves that schedulers based on these methods are work-conserving when the speedup is at least 2 and shows how they can be implemented using the concept of blocking flows. Section III shows how one can implement a practical distributed scheduler, based on one of the scheduling methods developed in Section II and evaluates its performance for speedups less than 2, using simulation. Section IV introduces a more sophisticated, scheduling method, shows that it too is work-conserving when the speedup is at least 2 and shows how it can be implemented using minimum cost blocking flows, in networks with convex cost functions. Section V describes a practical scheduler based on this method and evaluates it using simulation, showing that it can out-perform the simpler scheduler studied in section III. Section VI discusses a number of practical considerations for distributed scheduling and Section VII concludes the paper.

2. Work-Conserving Algorithms

In this section we describe a general scheduling strategy that can be used to obtain work-conserving scheduling algorithms for speedups of 2 or more. While these algorithms are not practical for real systems, they provide a conceptual foundation for other algorithms, that are practical.

For the purposes of analysis, we adopt a somewhat idealized view of the system operation (in Section VI, we discuss the implications of these assumptions for real systems). Specifically, we assume that the system operates in three discrete phases: an *arrival phase*, a *transfer phase* and a *departure phase*. During the arrival phase, each input receives up to T cells.[1] During the transfer phase, cells are moved from inputs to outputs, with each input constrained to send at most ST cells (S being the *speedup* of the system) and each output constrained to receive at most ST. During the output phase, each output sends up to T cells on its outgoing link. The scheduling algorithm determines which cells are transferred during each transfer cycle.

The scheduling strategy that we study in this section maintains an ordering of the non-empty VOQs at each input. The ordering of the VOQs can be extended to all the cells at an input. Two cells in the same VOQ are ordered according to their position in the VOQ. Cells in different VOQs are ordered according to the ordering of the VOQs. We say that a cell b *precedes* a cell c at the same input, if b comes before c in this cell ordering. For any cell c at an input, we let $p(c)$ be the number of cells at the same input as c that precede c and we let $q(c)$ be the number of cells at the output that c is going to.

We refer to a cell c as an ij-cell if it is at input i and is destined for output j. We say that a scheduling algorithm is *maximal* if during any transfer phase in which there is an ij-cell c that remains at input i, either input i transfers ST cells or output j receives ST cells. Given a method for ordering the cells at each input, we say that a scheduling algorithm is *ordered*, if in any transfer phase in which an ij-cell c remains at input i, either input i transfers ST cells that *precede* c or output j receives ST cells. Our scheduling strategy produces schedules that are maximal and ordered. We can vary the strategy by using different ordering methods. We describe two ordering methods that each lead to work-conserving scheduling algorithms. In fact, because there are many different maximal, ordered scheduling algorithms for any specific ordering method, we obtain two *families* of work-conserving scheduling algorithms.

For any cell c waiting at an input, we define the quantity $slack(c) = q(c) - p(c)$. For each of the methods studied, we'll show that $slack(c) \geq T$ at the start of each departure phase if $S \geq 2$. This implies that for any output with fewer than T cells in its outgoing queue, there can be no cells waiting in any input-side VOQs. This implies that the system is work-conserving.

[1] We assume throughout, that variable-length packets are segmented into fixed-length units for transmission through the interconnection network. We refer to these units as cells.

2.1. Batch CCF

The method we describe first is based on ideas first developed in the *Critical Cells First* method of [5]. Hence, we refer to it as the *Batch Critical Cells First* (BCCF) method. In the BCCF method, the relative ordering of two VOQs remains the same so long as they remain non-empty, but when a new VOQ becomes non-empty, it must be ordered relative to the others. When a cell c arrives and the VOQ for c's output is empty, we insert the VOQ into the existing ordering based on the magnitude of $q(c)$. In particular, if the ordered list of VOQs is $v_1, v_2, \ldots$, we place the VOQ immediately after the queue v_j determined by the largest integer j for which the number of cells in $v_1, \ldots, v_j$ is no larger than $q(c)$. Notice that this ensures that $slack(c)$ is non-negative right after c arrives. A specific scheduling algorithm is an instance of the BCCF method if it produces schedules that are maximal and ordered with respect to this VOQ ordering method. To show that $slack(c) \geq T$ at the start of each departure phase, we need two lemmas.

Lemma 1. For a system using the BCCF method, if c is any cell that remains at its input during a transfer phase, then $slack(c)$ increases by at least ST during the transfer phase.

proof. Since the VOQ ordering does not change during a transfer phase (more precisely, VOQs that remain non-empty during the transfer phase have the same relative order), any maximal, ordered scheduling algorithm either causes $q(c)$ to increase by ST or causes $p(c)$ to decrease by ST. In either case, $slack(c)$ increases by ST. ■

Note that as long as a cell c remains at an input, each arrival phase and departure phase cause $slack(c)$ to decrease by at most T. So, if $S \geq 2$, $slack(c)$ cannot go down over the course of a complete time step, comprising an arrival phase, a transfer phase and a departure phase.

Lemma 2. For a system using the BCCF method with $S \geq 2$, if c is any cell at an input just before the start of the departure phase, then $slack(c) \geq T$.

proof. We show that for any cell c present at the end of an arrival phase, $slack(c) \geq -T$. The result then follows from Lemma 1 and the fact that $S \geq 2$. The proof is by induction on the time step.

For any cell c that arrives during the first time step, $p(c) \leq T$ at the end of the arrival phase, so $slack(c) \geq -T$ at the end of the arrival phase. Since $S \geq 2$, there can be no net decrease in $slack(c)$ from one time step to the next, so $slack(c)$ remains $\geq -T$ at the end of each subsequent arrival phase, so long as c remains at the input.

If a cell c arrives during step t and its VOQ is empty when it arrives, then the rule used to order the VOQ relative to the others ensures that $slack(c) \geq 0$ right after it arrives. Hence, $slack(c) \geq -T$ at the end of the arrival phase and this remains true at the end of each subsequent arrival phase, so long as c remains at the input.

If a cell c arrives during step t and its VOQ is not empty, but was empty at the start of the arrival phase, then let b be the first arriving cell to be placed in c's VOQ during this arrival phase. Then, $slack(b)$ was at least 0 at the time it arrived and at most $T-1$ cells can have arrived after b did in this arrival phase. If exactly r of these precede b, then at the end of the arrival phase,

$$\begin{aligned} slack(c) &\geq slack(b) - ((T-1) - r) \\ &\geq (-r) - ((T-1) - r) \\ &\geq -T \end{aligned}$$

If a cell c arrives during step t and its VOQ was not empty at the start of the arrival phase, then let b be the last cell in c's VOQ at the start of the arrival phase. By the induction hypothesis, $slack(b) \geq -T$ at the end of the previous arrival phase. Since the subsequent transfer phase increases $slack(b)$ by at least $2T$ and the departure phase decreases it by at most T, $slack(b) \geq 0$ at the start of the arrival phase in step t. During this arrival phase, at most T new cells arrive at c's input. Let r be the number of these arriving cells that precede b. Then at the end of the arrival phase

$$\begin{aligned} slack(c) &\geq slack(b) - ((T-1) - r) \\ &\geq (-r) - ((T-1) - r) \\ &\geq -T \end{aligned}$$

Hence, $slack(c) \geq -T$ at the end of the arrival phase in all cases and this remains true at the end of each subsequent arrival phase, so long as c remains at the input. ■

Lemma 2 leads immediately to a work-conservation theorem for BCCF.

Theorem 1. For $S \geq 2$, any scheduler using the BCCF method is work-conserving.

2.2. Batch LOOFA

Our second algorithm is based on ideas first developed in the *Least Occupied Output First* algorithm of [7], so we refer to it as the *Batch Least Occupied Output First* (BLOOFA) algorithm. In the BLOOFA algorithm, the VOQs are ordered according to the number of cells in the output-side queues. VOQs going to outputs with fewer cells precede VOQs going to outputs with more cells. Outputs with equal numbers of cells are ordered by the numbering of the outputs. We define the BLOOFA algorithm to be the combination of this VOQ ordering method with any maximal, ordered scheduling algorithm. We show that $slack(c) \geq T$ at the start of each departure phase, using the same overall strategy used for BCCF. As before, we need two lemmas. The arguments are similar, but complicated by the fact that the relative ordering of VOQs can change during a transfer phase.

Lemma 3. For a system using the BLOOFA method, during a transfer phase, the minimum slack at any input that does not transfer all of its cells during the transfer phase, increases by at least ST.

proof. Let c be any cell at input i, and let j be the output that c is going to. Let *minSlack* be the smallest value of the slack at input i just before the transfer phase, and let $slack(c) = minSlack + \sigma$. We will show that $slack(c)$ increases by at least $ST - \sigma$ during the transfer phase. The lemma then follows directly. (Note that it is not sufficient to prove that the slack of a cell c that has minimum slack at the start of the transfer phase increases by ST, since c may not be a cell of minimum slack at the end of the transfer phase.)

We say that a cell b at input i *passes* c, if before the transfer phase, c precedes b and after the transfer phase b precedes c. If no cells pass c during the transfer phase, then by the definition of maximal, ordered scheduling algorithms, either $q(c)$ increases by ST or $p(c)$ decreases by ST. In either case, $slack(c)$ increases by at least $ST \geq ST - \sigma$.

Assume then, that there are $r>0$ cells that pass c and let b be the cell in the set of cells that pass c that comes latest in the cell ordering (before the transfer phase). For clarity, let $q_0(x)$ denote the value of $q(x)$ before the transfer phase and let $q_F(x)$ denote the value of $q(x)$ after the transfer phase. Similarly for the functions p and *slack*.

Let m be the number of cells received by output j during the transfer and let k be the number of cells that precede b before the transfer, but do not precede c. Then,

$$q_0(c) + m = q_F(c) \geq q_F(b) \geq q_0(b)$$

and $p_0(b) = p_0(c) + k$. Now,

$$\begin{aligned} q_0(c) - p_0(c) &= minSlack + \sigma \\ &\leq q_0(b) - p_0(b) + \sigma \\ &\leq (q_0(c) + m) - (p_0(c) + k) + \sigma \end{aligned}$$

So $(m-k) \geq -\sigma$. Since b passes c, its output must receive fewer than m cells during the transfer phase, so ST cells that precede it at the start of the transfer phase must be forwarded. Of these at least $ST-(k-r)$ must also precede c at the start of the phase. So,

$$p_F(c) \leq p_0(c) + r - (ST - (k - r)) \leq p_0(c) - ST + k$$

Combining this, with $q_F(c) = q_0(c) + m$ gives,

$$\begin{aligned} slack_F(c) &= q_F(c) - p_F(c) \\ &\geq (q_0(c) + m) - (p_0(c) - ST + k) \\ &\geq slack_0(c) + ST + (m - k) \\ &\geq slack_0(c) + ST - \sigma \end{aligned}$$

That is, $slack(c)$ increases by at least $ST - \sigma$. ■

Note that each arrival phase causes $slack(c)$ to decrease by at most T. However, it is not so easy to bound the decrease in $slack(c)$ during the departure phase. The source of the difficulty is that other cells at c's input can pass it during the departure phase, making it hard to bound the overall change in $slack(c)$. However, if $slack(c)$ is at least T before the departure phase begins, then $q(c)$ must also be at least T. This means that T cells will depart from c's output, making it impossible for other cells at c's input to pass c. Thus, if $slack(c)$ is at least T before the departure phase, then $slack(c)$ is at least 0 after the departure phase. It turns out that this is sufficient to establish that BLOOFA is work-conserving when $S \geq 2$.

Lemma 4. For a system using the BLOOFA method with $S \geq 2$, if c is any cell at an input just before the start of the departure phase, then $slack(c) \geq T$.

proof. We show that for any cell c present at the end of the arrival phase, $slack(c) \geq -T$. The result then follows from Lemma 3 and the fact that $S \geq 2$. The proof is by induction on the time step.

For any cell c that arrives during the first time step, $p(c) \leq T$ at the end of the arrival phase, so $slack(c) \geq -T$ at the end of the arrival phase. Since $S \geq 2$, Lemma 3 implies that $slack(c) \geq T$ at the end of the transfer phase, if it is still present at the input. By the discussion just before the statement of Lemma 4, this means that $slack(c) \geq 0$ following the departure phase, which in turn means that $slack(c) \geq -T$ at the end of the next arrival phase. This remains true at the end of every subsequent arrival phase until c is transferred to the output.

Suppose then, that c arrives during step t. If, at the end of the arrival phase, the only cells that precede c also arrived during step t, then $slack(c) \geq -T$ at the end of the arrival phase. By the argument at the end of the last paragraph, this remains true at the end of every subsequent arrival phase until c is transferred to the output.

If at the end of the arrival phase in step t, there are cells that precede c that were present at the start of the arrival phase, then let b be the cell in this set of cells that does not precede any of the others in the set. Because b arrived before step t, $slack(b) \geq -T$ at the end of the previous arrival phase, by the induction hypothesis. This implies that $slack(b) \geq 0$ at the start of the arrival phase in step t. Let k be the number of cells that arrive during the arrival phase that precede b at the end of the arrival phase. Let m be the number of cells that arrive during the arrival phase that precede c but not b at the end of the arrival phase. Since $k + m \leq T$ and $slack(b) \geq -k$,

$$\begin{aligned} slack(c) = q(c) - p(c) &\geq q(b) - (p(b) + m) \\ &= slack(b) - m \geq -(m + k) \geq -T \end{aligned}$$

This remains true at the end of each subsequent arrival phase, so long as c remains at the input. ■

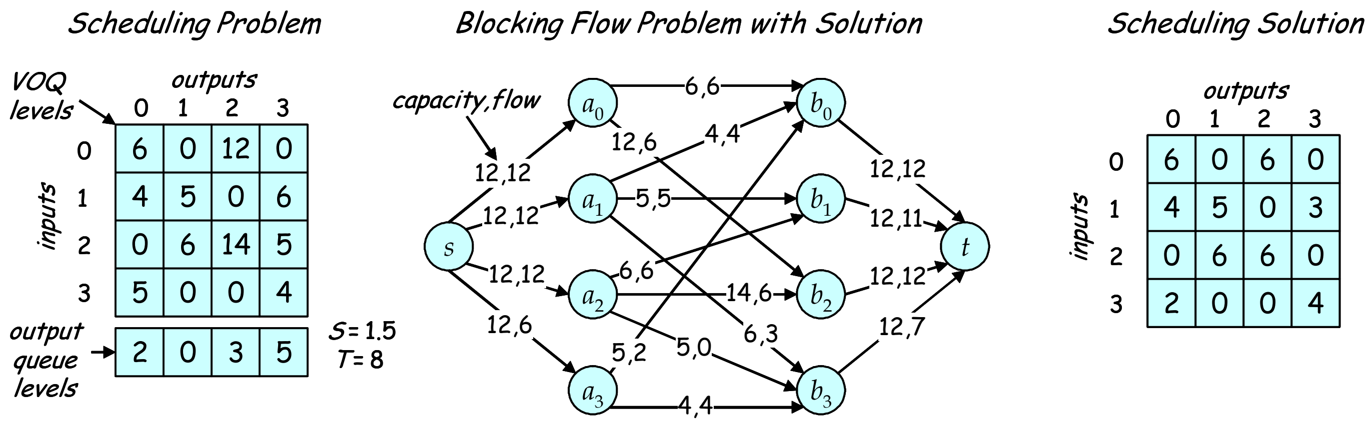

Figure 1. Example showing a maximal ordered schedule constructed from a blocking flow.

Lemma 4 leads immediately to the work-conservation theorem for BLOOFA.

Theorem 2. For $S \geq 2$, any scheduler using the BLOOFA method is work-conserving.

2.3. Implemention of Maximal, Ordered Schedulers

We have shown that the combination of two different VOQ ordering strategies with a maximal, ordered scheduling algorithm ensures work-conserving operation when the speedup is at least 2. We now need to show how to realize a maximal, ordered scheduling algorithm. We start with a centralized algorithm and then show how it can be converted into an iterative, distributed algorithm. While the overhead of such iterative algorithms makes them impractical, they provide the basis for non-iterative algorithms that are practical.

The key observation is that the scheduling problem is equivalent to finding a *blocking flow* in an acyclic flow network [13]. A flow network is a directed graph with a distinguished source vertex s, a distinguished *sink* vertex t and a non-negative *capacity* for each edge. A flow, in such a network, is a non-negative function defined on the edges. The flow on an edge must not exceed its capacity and for every vertex but s and t, the sum of the flow values on the incoming edges must equal the sum of the flow values on the outgoing edges. An edge in the network is called *saturated*, if the flow on the edge is equal to its capacity. A blocking flow is one for which every path from s to t contains at least one saturated edge. (Note that a blocking flow is not necessarily a maximum flow.)

To convert the scheduling problem to the problem of finding a blocking flow, we first need to construct a flow network. Our network has a source s, a sink t, n vertices referred to as inputs and another n vertices referred to as outputs. There is an edge with capacity ST from s to each input. Similarly, there is an edge with capacity ST from each output to t. For each non-empty VOQ at input i of the router with cells for output j, there is an edge in the flow network from input i to output j with capacity equal to the number of cells in the VOQ. (An example of a flow network constructed to solve a particular scheduling problem together with the corresponding solution is shown in Figure 1.)

For any integer flow, we can construct a schedule that transfers cells from input i to output j based on the flow on the edge from input i to output j. Note that such a schedule does not violate any of the constraints on the number of cells that can be sent from any input or to any output. Also note that any blocking flow corresponds to a maximal schedule, since any blocking flow corresponding to a schedule which fails to transfer a cell c from input i to output j cannot saturate the edge from input i to output j, hence it must saturate the edge from s to i or the edge from j to t. Such a flow corresponds to a schedule in which either input i sends ST cells or output j receives ST.

Dinic's algorithm [13] for the *maximum flow problem* constructs blocking flows in acyclic flow networks as one step in its overall execution. There are several different variants of Dinic's algorithm, that use different methods of constructing blocking flows. The most straightforward method is to repeatedly search for st-paths with no saturated edges and add as much flow as possible along such paths. We can obtain a maximal, ordered scheduler by modifying Dinic's algorithm so that it preferentially selects edges between input vertices and output vertices, according to the VOQ ordering at the input. The blocking flow shown in Figure 1 was constructed in this way, based on the BLOOFA ordering.

If paths are found using depth-first search and edges leading to dead-ends are removed as they are discovered, Dinic's algorithm finds a blocking flow in $O(mn)$ time where m is the number of edges and n is the number of vertices in the flow graph. Because the flow graphs corresponding to schedules have bounded depth and because the number of inputs, outputs and edges are all bounded by the number of non-empty VOQs, the algorithm finds a blocking flow in these graphs in $O(v)$ time where v is the number of non-empty VOQs. This yields an optimal centralized scheduling algorithm. However, since v can be as large as n^2 (where n is the number of inputs of the interconnection network), this is not altogether practical.

We can obtain a distributed, iterative scheduling algorithm based on similar ideas. Rather than state this in the language of blocking flows, we describe it directly as a scheduling algorithm. In the distributed scheduler, we first have an exchange of messages in which each output announces the number of cells in its outgoing queue. The inputs use this information to maintain their VOQ order. Note that this requires that each output send n messages and each input receive n messages. Next, the inputs and outputs proceed through a series of rounds.

In each round, the inputs that have uncommitted cells to send and have not yet committed to sending ST cells, send *bid* messages to those outputs that are still prepared to accept more cells. The inputs construct their bids in accordance with the VOQ ordering. In particular, an input commits all the cells it has for the first output in the ordering and makes similar maximal bids for subsequent outputs until it has placed as many bids as it can. Inputs may not *overbid*, as they are obliged to send cells to any output that accepts a bid. Note that at most one of the bid messages an input sends during a round does not commit all the cells that it has for the target output.

During each round, outputs receive bids from inputs and accept as many as possible. If an output does not receive bids for at least ST cells, it does nothing during this round. In particular, it sends no message back to the inputs. Such a "response" is treated by the bidding inputs as an *implicit accept* and is taken into account in subsequent bids. Once an output has received bids for a total ST cells, it sends an accept message to all the inputs (not just those that sent it bids). The accept message contains a pair of values (i,x) and it means that the output accepts all bids received from inputs whose index is less than i, rejects all bids from inputs whose index is greater than i and accepts exactly x cells from input i. Once an output sends an accept message, its role in the scheduling is complete.

This procedure has some attractive properties. First, note that each output sends n messages in the bidding process, so each input receives only n messages from outputs. Also, an input sends at most two bids to any particular output, so an input sends at most $2n$ bids and an output receives at most $2n$ bids. Thus, the number of cells that must be handled at any input or output during the scheduling is $O(n)$. Unfortunately, this does not imply that the algorithm runs in $O(n)$ time, since it can require up to n rounds and some outputs may have to handle close to n messages during most rounds.

It's possible to reduce the time for each round by having the switch elements that make up the interconnection network participate in the handling of bids and responses. However, in the next section we turn our attention instead, to algorithms that are simpler to implement and which, while not provably work-conserving, are able to match the performance of the work-conserving algorithms, even under extreme traffic conditions.

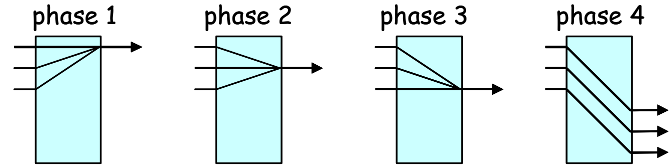

Figure 2. Typical stress test

3. Distributed BLOOFA

The work-conserving algorithms discussed in the previous section can be implemented using iterative algorithms that require a potentially large number of message exchanges. In this section, we formulate a distributed algorithm that approximates the behavior of the BLOOFA algorithm while requiring just one exchange of messages. Our *Distributed BLOOFA* (DBL) algorithm avoids the need for many message exchanges by having the inputs structure their bids to avoid the situation where some outputs are swamped with more bids than they can accept, while others are left with no bids. Specifically, the inputs use a technique introduced in [11] called *backlog-proportional allocation* to limit the number of bids that are made for any output.

DBL starts with each input i sending a message to each output j, telling it how many cells $B(i,j)$ it has in its VOQ for output j. Each output j then sends a message to all inputs containing the number of cells in its output queue ($B(j)$) and the total number of cells that inputs have to send it ($B(+,j)$[2]). Note that each input and output sends and receives n messages. Once this exchange of messages has been made, each input independently decides how many cells to send to each output. To prevent too many cells from being sent to any output, input i is allowed to send at most $ST \times B(i,j)/B(+,j)$ cells to output j. Each input then orders the outputs according to the length of their output queues and goes through this list, assigning as many cells as it is permitted for each output, before going to the next output in the list. The scheduling is complete when the input has assigned a total of ST cells or has assigned all the cells permitted by the bound.

We studied the performance of DBL using simulation for speedups between 1 and 2. We start with an adversarial traffic pattern, we call a *stress test*, that is designed to probe the limits of the algorithm's performance. The stress test consists of a series of phases, as illustrated in Fig. 3. In the first phase, the arriving traffic at each of several inputs is directed to a single output. This causes each of the inputs to build up a backlog for the target output. The arriving traffic at all the inputs is then re-directed to a second output, causing the accumulation of a backlog for the second output. Successive phases proceed similarly, creating backlogs at each input for each of several outputs. During the last phase, the arriving traffic at all inputs

2. We use the addition symbol ('+') as a function argument to denote the summation of the function over all values of that argument.

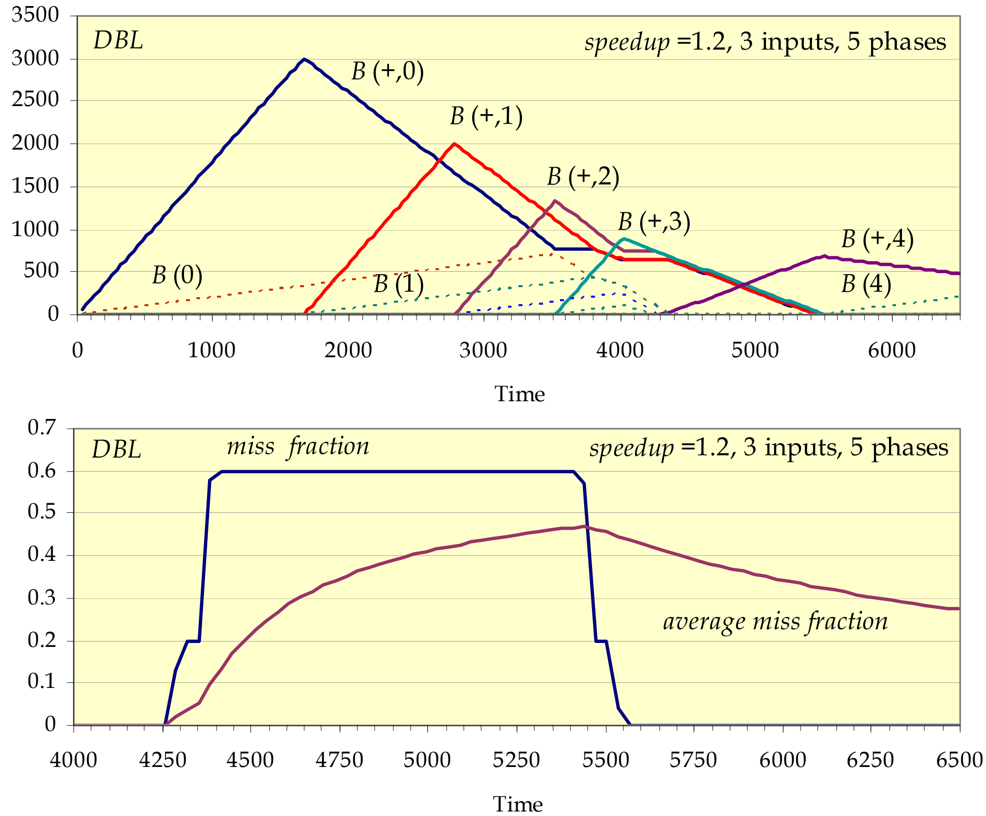

Figure 3. Results from sample stress test for distributed BLOOFA - buffer levels (top) and miss fraction (bottom).

is re-directed to a distinct new output for each input. Since each of the target outputs of the last phase has only a single input directing traffic to it, that input must send cells to it as quickly as they come in, while simultaneously clearing the accumulated backlogs for the other outputs, in time to prevent underflow at those other outputs. This creates an extreme condition that can lead to underflow. The timing of the transitions between phases is chosen so that the total number of cells in the system directed to each output is approximately the same at the time the transition takes place. The stress test can be varied by changing the number of participating inputs and the number of phases.

Figure 3 shows results from a sample stress test. The top chart shows the buffer levels in the VOQs at input 0 and the buffer levels at outputs 0 to 4 (by symmetry, the VOQ lengths at other inputs will be approximately the same as those at input 0). The time unit is the update interval, T. The unit of storage is the number of cells that can be sent on an external link during the update interval. Notice that during the last phase of the stress test $B(0,4)$ rises, indicating that input 0 is unable to transfer cells to output 4 as quickly as they come in. This results in loss of link capacity at output 4. The second chart shows the *miss fraction* at output 4 during this last phase. The term "*miss*" refers to a missed opportunity to send a cell. The miss fraction measures the fraction of the link capacity that is effectively lost during the last phase due to such misses and is a measure of how far the system deviates from being work-conserving. The curve labeled simply, "*miss fraction*" measures the average miss fraction during successive measurement intervals (the measurement intervals are 25 time units long). The curve labeled "*average miss fraction*" is the fraction of the link capacity lost from the start of the last phase to the time plotted. We observe that almost 30% of the link's capacity is effectively lost between the start of the last phase and the end of the period shown.

The left-hand chart in Figure 4 shows how DBL performs on a series of stress tests with speedups varying between 1 and 1.5. (In these tests, the length of the stress test was set to 1.2 times the length of time that would be required to forward all the cells received during the first phase in an ideal output-queued switch. We see here that the average miss fraction (for the output targeted by input 0 in the last phase) drops steadily with increasing speedup, dropping to zero before the speedup reaches 1.5. We performed 90 sets of stress tests, using different numbers of inputs and phases (up to 15 inputs and 15 phases). The results plotted in the figure are the worst-cases for 2, 3, 4 and 5 inputs. In all cases, the average miss fraction for the last phase target output dropped to zero for speedups greater than 1.5.

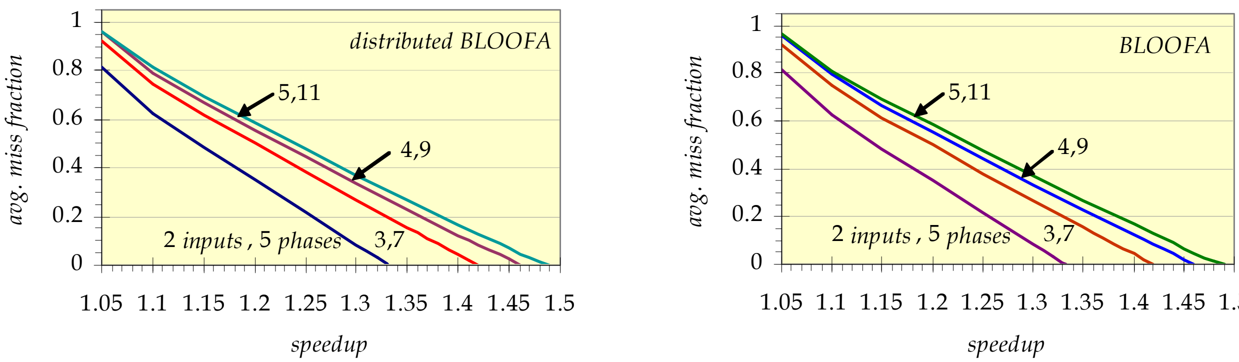

Figure 4. Miss fraction for DBL and BLOOFA on a variety of stress tests

To compare DBL to BLOOFA, we performed the same series of 90 stress tests on BLOOFA. For speedups below 2, the method used to select which inputs send traffic to a given output can have a significant effect on the performance of BLOOFA. For the results given here, we went through the outputs in order (from smallest output-side backlog to largest) and for each output *j*, we assigned traffic from the different inputs to output *j* in proportion to the fraction that each could supply of the total that all inputs could send to *j* in this update interval. The right-hand chart in Figure 4 shows the results of these stress tests on BLOOFA. Although a close examination of the results reveal small differences between the distributed and centralized versions of BLOOFA, the results are virtually indistinguishable. We find that the approximation introduced by using the backlog-proportional allocation method to enable efficient distributed scheduling, has a negligible effect on the quality of the scheduling results, even though the distributed version is not known to be provably work-conserving for any speedup.

We have also studied the performance of DBL for less adversarial (although, still very demanding) traffic conditions. In particular, we have studied bursty traffic situations in which there is one output (referred to as the *subject* output), for which traffic is arriving continuously at a specified fraction of the link rate. The input at which the subject's traffic arrives changes randomly as the simulation progresses (it remains with a given input for an exponentially distributed time interval). Each of the inputs that is not currently providing traffic for the subject has its own *target* output (not equal to the subject) to which it sends traffic, changing targets randomly and independently of all other inputs (an input retains its current target for an exponentially distributed time interval). With this traffic pattern, roughly one fourth of the outputs that are not the subject are overloaded at any one time (they are targets of two or more inputs). An ideal scheduler will forward cells to the subject output as fast as they come in, preventing any input-side queueing of cells for the subject. However, the other outputs can build up significant input side backlogs (due to the transient overloads they experience), leading to contention that can affect the subject output. Figure 5 shows an example of what can happen in a system subjected to this type of traffic. The top chart shows the amount of data buffered for the subject output (which is output 0) at all inputs ($B(+,0)$), the amount of data buffered at the input, which is currently receiving traffic for the subject ($B(i,0)$) and the amount of data buffered at the subject ($B(0)$). The unit of storage is the amount of data received on an external link during an update interval and the time unit is the update interval. The discontinuities in the curve for $B(i,0)$ occur when the input that is currently receiving traffic for the subject changes (i.e., the value of i changes). The bottom chart shows the instantaneous value of the miss fraction.

Figure 6 shows the average miss fraction from a large number of bursty traffic simulations with varying input load and speedup. It's interesting to note that the miss fraction reaches its peak when the input load is between 0.8 and 0.9. Larger input loads lead to a sharp drop in the miss fraction. The explanation for this behavior is that when the input load approaches 1, output-side backlogs tend to persist for a long period of time and it's only when the output-side backlogs are close to zero that misses can occur. As one would expect, the miss fraction drops quickly as the speedup increases. Note that for speedup 1.15 the miss fraction never exceeds 2%, meaning that only a small fraction of the link capacity is lost.

It should be noted that the bursty traffic model used in these studies represents a very extreme situation. A more realistic bursty traffic model would have a large number of bursty sources (at least a few tens) with more limited peak rates sharing each input link (at least a few tens of sources per link). Such a model is significantly less bursty than the one used here.

4. The Output Leveling Algorithm

The intuition behind the BLOOFA algorithm is that by favoring outputs with smaller queues, we can delay the possibility of underflow and potentially avoid that possibility altogether. Theorem 2 tells us that for a speedup of 2 or more, we can avoid underflow, but it does not say anything about what happens with smaller speedups. When there are several output queues of nearly the same length, BLOOFA transfers as many

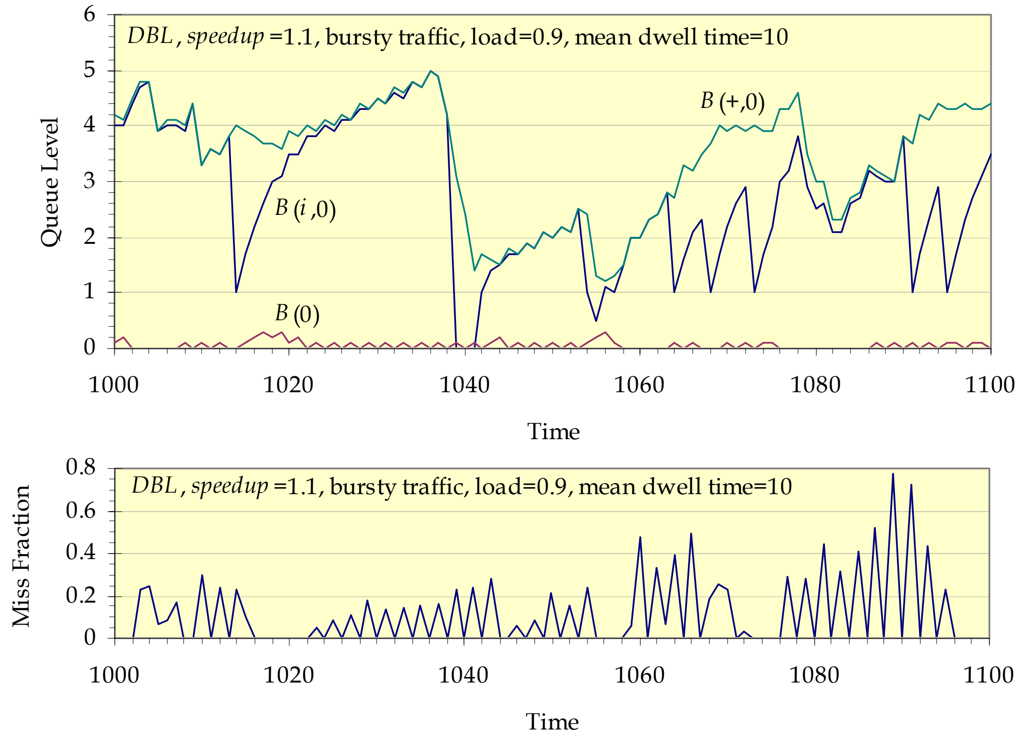

Figure 5. Time series showing performance of DBL for bursty traffic.

cells as possible to the shortest queues, potentially preventing any cells from reaching slightly longer queues. It seems likely that we could get better performance by balancing the transfers so that the resulting output queue lengths are as close to equal as possible. This is the intuition behind the *Output Leveling Algorithm* (OLA), which we consider next. In this section we show that OLA, like BCCF and BLOOFA is work-conserving for speedups of 2 or more. Subsequently, we study the performance of OLA and a practical variant of OLA and show that these algorithms can out-perform BLOOFA and DBL.

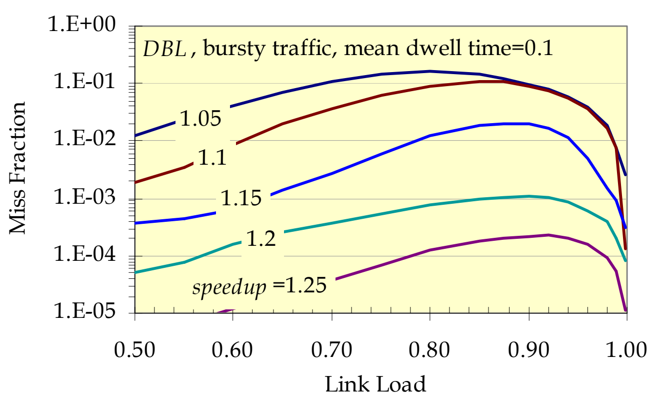

Figure 6 Performance of DBL on bursty traffic with varying speedups and subject, target dwell times

OLA orders cells at an input in the same way that BLOOFA does. Let $B(i,j)$ and $B(j)$ be the lengths of the VOQs and output queues respectively, immediately before a transfer phase and let $x(i,j)$ be the number of cells transferred from input i to output j during the transfer. We say that the transfer is *level* if for any pair of outputs j_1 and j_2,

$$B(j_1) + x(+,j_1) < B(j_2) + x(+,j_2) - 1$$

implies that $x(+, j_1) = \min\{ST, B(+,j_1)\}$. We now define OLA as any scheduling algorithm that produces schedules that are maximal and level.

4.1. Work Conservation

We will use essentially the same strategy as before to show that OLA is work-conserving when the speedup is at least 2. However, to show that the minimum slack increases by ST at each input during a transfer phase, we first need to show how a transfer phase scheduled by OLA can be decomposed into a sequence of *sub-phases*. Let $B(i,j)$ and $B(j)$ be the lengths of the VOQs and output queues respectively, immediately before a transfer phase and let $x(i,j)$ be the number of cells transferred from input i to output j during the transfer. Each of the sub-phases corresponds to the transfer of up to one cell from each input and up to one cell to each output. We let $x_k(i,j)$ denote the number of cells transferred from input i to output j by the first k sub-phases. At the end of sub-phase k, the outputs are ordered in increasing order of $B(j) + x_k(+,j)$ with ties broken according to the output numbers. The ordering of the outputs is used to order the VOQs at each input and this ordering is extended to all the cells at each input. We say that a cell b precedes a cell c following sub-phase k if b comes before c in this cell ordering. We define $q_k(c)=B(j)+x_k(+,j)$ and we define $p_k(c)$ to be the number of cells at c's input that precede it in the ordering at the end of sub-phase k. We also define $slack_k(c)=q_k(c)-p_k(c)$. Let $slack_0(c)$ be the value of $slack(c)$ before the transfer phase begins and note that if k is the last sub-phase, then $slack_k(c)$ is equal to the value of $slack(c)$ following the transfer phase.

Given a schedule constructed by an OLA scheduler, we construct sub-phases iteratively. To construct sub-phase k, by repeating the following step until there are no outputs that are eligible to be selected.

> Select an output j that has not yet been selected in this sub-phase for which $x_{k-1}(+,j)<x(+,j)$ and which, among all such outputs, has the minimum value of $q_{k-1}(c)$. If there are multiple outputs that satisfy this condition, select the output that comes first in the fixed numbering of the outputs. Then, select some input i that has not yet been selected in this sub-phase for which $x_{k-1}(i,j)<x(i,j)$. If there is such an input, include the transfer of a cell from input i to output j in sub-phase k, making $x_k(i,j)=x_{k-1}(i,j)+1$.

We will use this decomposition to show that the minimum slack at each input increases by at least ST during each transfer phase.

Lemma 5. For a system using the OLA method, during a transfer phase, the minimum slack at any input that does not transfer all of its cells during the transfer phase, increases by at least ST.

proof. Because OLA constructs maximal schedules, any transfer phase that leaves cells at input i must either transfer ST cells from input i or must transfer ST cells to every output j for which a cell remains at input i following the transfer phase. This means that if we decompose the transfer phase into sub-phases, as described above, there will be at least ST sub-phases. We show below that every one of these sub-phases increases the minimum slack at input i. Hence, the minimum slack increases by ST over the complete transfer phase.

Let k be the index of any sub-phase and let c be any cell at input i which is not transferred during sub-phase k and for which $slack_{k-1}(c)$ is minimum among all cells at input i. Let j be the output that c is going to. If output j receives no cell during sub-phase k, then input i must transfer a cell during sub-phase k. The selection rule used to construct sub-phases ensures that the transferred cell precedes c. Hence, $p_k(c) = p_{k-1}(c)-1$ and thus, $slack_k(c) = slack_{k-1}(c)+1$.

If output j does receive a cell, then $q_k(c) = q_{k-1}(c)+1$. If no cell at input i passes c during the sub-phase, then $slack_k(c) \geq slack_{k-1}(c)+1$. Suppose then, that there is one or more cell that passes c during the sub-phase and let d be such a cell. Since c precedes d before the sub-phase $q_{k-1}(c)\leq q_{k-1}(d)$ and $p_{k-1}(c)<p_{k-1}(d)$. Since d precedes c after the sub-phase, no cell is received by d's output during the sub-phase and so $q_{k-1}(d) \leq q_{k-1}(c)+1$. Because $slack_{k-1}(c)\leq slack_{k-1}(d)$, $p_{k-1}(d)-p_{k-1}(c)\leq q_{k-1}(d)-q_{k-1}(c)\leq 1$ which means that there are no cells that fall between c and d in the cell ordering. This implies that d is the only cell that passes c during the sub-phase. Because d's output receives no cell during the sub-phase, there must be some cell that precedes d that is transferred from input i during the sub-phase and this cell must also precede c. Thus, $p_k(c) = p_{k-1}(c)$ and so $slack_k(c) = slack_{k-1}(c)+1$. ■

As before, we note that each arrival phase causes $slack(c)$ to decrease by at most T. Also, as before, if $slack(c)$ is at least T before the start of a departure phase, then $slack(c)$ is at least zero, after the departure phase. This is sufficient to establish that OLA is work-conserving when $S \geq 2$.

Lemma 6. For a system using the OLA method with $S \geq 2$, if c is any cell at an input just before the start of the departure phase, then $slack(c) \geq T$.

The proof of Lemma 6 is just like the proof of Lemma 4, except that it uses Lemma 5, in place of Lemma 3. Lemma 6 leads immediately to the work-conservation theorem for OLA.

Theorem 3. For $S\geq 2$, any scheduling algorithm using the OLA method is work-conserving.

4.2. Implementating OLA

An OLA scheduler can be implemented exactly either using linear programming or by solving a minimum cost, maximum flow problem with a convex cost function. We outline the latter approach, as it serves to motivate more practical, approximate variants.

In the classical version of the minimum cost, maximum flow problem [1,13], each edge has an associated cost coefficient, which is multiplied by the flow on the edge to get the edge's contribution to the overall cost of the flow. There are several well-known efficient algorithms for solving the minimum cost, maximum flow problem. Interestingly, these algorithms can be generalized to handle networks in which the cost is a convex function of the flow on the edge, rather than a linear function (x^2 is an example of a convex function).

The OLA scheduling algorithm can be reduced to solving a minimum cost, maximum flow problem with a convex edge cost function. An example of such a reduction is shown in Figure 7, along with a solution and the corresponding schedule. The flow graph is constructed in the same way as was discussed in Section 2. The only difference is the introduction of non-zero costs on the edges from the output vertices to the sink vertex t. The cost of an edge from output j to t carrying a flow of magnitude x is defined as $C(x) = (x + B(j))^2$. A minimum cost, maximum flow for this network corresponds directly to an OLA schedule. The convexity of the cost function ensures that the flows on different output to sink edges

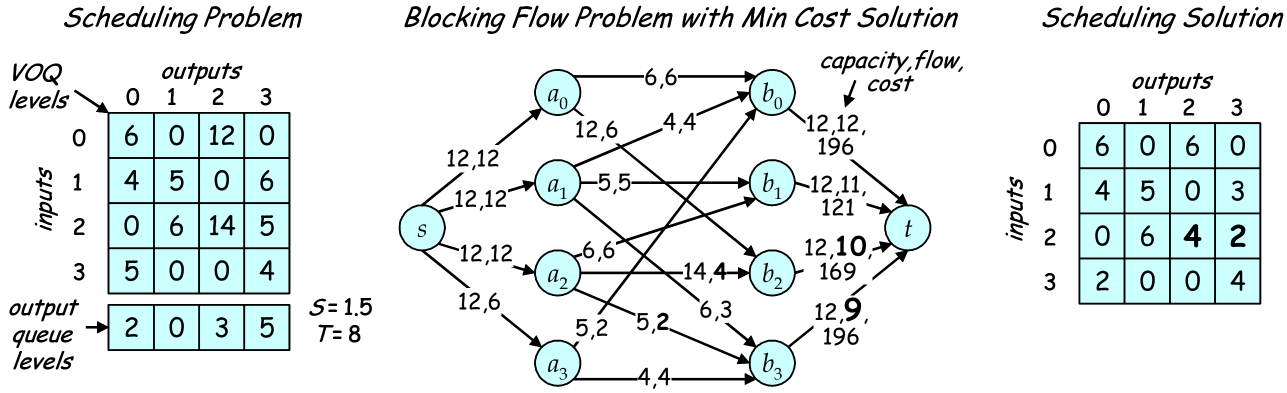

Figure 7. Implementing OLA using minimum-cost blocking flow with convex cost function. Differences from earlier solution highlighted in **bold**.

result in costs that are as nearly equal as the various edge capacities allow (if a flow can be shifted from a higher cost edge to a lower cost edge, there is a net reduction in cost, because the lower cost edge has lower incremental cost, per unit flow). The use of the offset $B(j)$ in the edge cost means that the costs of the flows on two output-to-sink edges are equal whenever the corresponding schedules yield equal levels at the output queue. Reference [1] describes an algorithm that finds a minimum cost, maximum flow in $O((m \log K)(m + n \log n))$ time on an arbitrary network with n vertices, m edges and maximum edge capacity K. While this algorithm is not useful for distributed scheduling in real systems, it can be used in performance studies to establish a benchmark for more practical algorithms that seek to approximate the behavior of OLA.

5. Distributed OLA

We start by describing an approximate centralized version of OLA. We then show how this can be converted to a distributed scheduler, using an extension of the backlog-proportional allocation method introduced earlier.

Our approximate centralized algorithm uses an array $x(i,j)$ which is initialized to zero and which defines the number of cells to be transferred from input i to output j, when the scheduling algorithm completes. It also uses a parameter $\Delta \le ST$, which determines the accuracy of the approximation. During its execution, the algorithm maintains a list of the outputs, sorted in increasing order of $x(+,j) + B(j)$. The algorithm repeats the following step so long as there are at least two outputs on the list.

> Let j_1 and j_2 be the indices of the first two outputs on the list. Increase $x(+,j_1)$, by repeatedly increasing $x(i,j)$ for selected values of i (input selection criteria are discussed below). Stop when $x(+,j_1) + B(j_1) = x(+,j_2) + B(j_2) + \Delta$, or when $x(+,j_1) = ST$ or when $x(+,j_1) = B(+,j_1)$, whichever occurs first. If either of the last two conditions occurs, remove j_1 from the list. Otherwise, move it down the list so as to maintain the ordering criterion.

When the list has been reduced to a single output j, the algorithm increases $x(+,j)$ until $x(+,j) = \min\{ST, B(+,j)\}$ or until all inputs with cells for output j have scheduled all they can (ST).

The number of steps performed by the algorithm is at most nST/Δ. It can be implemented to run in $O(m + (ST/\Delta)n^2)$ time, where m is the number of non-empty VOQs. This can be improved to $O(m + (ST/\Delta)n \log n)$, if the list is replaced with a heap. If Δ=1, the algorithm computes an OLA schedule (regardless of the input selection criterion). For larger values of Δ, it implements a Δ-OLA schedule, which is defined as any maximal schedule for which

$$B(j_1) + x(+,j_1) < B(j_2) + x(+,j_2) - \Delta$$

implies that $x(+,j_1) = \min\{ST, B(+,j_1)\}$. That is, a Δ-OLA scheduler allows the output queue differences at the end of a transfer phase to exceed Δ, only if there is no way to transfer more cells to the outputs with the smaller queues. Δ-OLA schedulers, like OLA schedulers are work-conserving when the speedup is at least 2 (a slight variant of the proof used for OLA can be used to show this). For smaller speedups, we can tradeoff scheduling performance against running time by adjusting Δ.

The criterion used to select the next input to use to effect an increase in $x(+,j_1)$ does not affect the work-conservation condition. However, different choices can affect performance when the speedup is less than two. In the performance results reported below, we distribute the load approximately evenly among all inputs with traffic for output j_1, using a round-robin technique. We maintain a list of inputs that can still send to j_1 (they have both cells for j_1 and uncommitted bandwidth) and use the first input on the list to increase the flow to j_1. To obtain an even distribution, we take at most Δ from an input at a time and then move that input to the end of the list. This method can be implemented without increasing the time complexity of the algorithm.

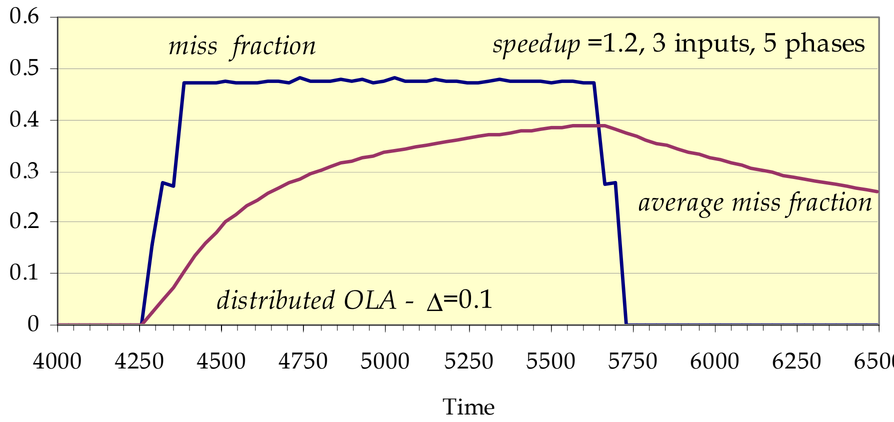

Figure 8. Example stress test for distributed OLA

To convert a Δ-OLA scheduler to a practical distributed scheduler, we use the *backlog proportional allocation* technique introduced earlier to allow inputs to divide the responsibility for supplying traffic to the different outputs. This allows each input to operate independently of the others, once the initial exchange of information takes place. As with DBL, this initial exchange supplies input i with the values of $B(j)$ and $B(+,j)$ for every output j. Input i also has the values $B(i,j)$ for all j and it uses these to compute values $\sigma(i,j) = B(i,j)/B(j)$. Given this information, input i makes its scheduling decisions in a way that is similar to the centralized algorithm. In particular, input i maintains a list of the outputs for which it has cells, sorted in increasing order of $B(j) + x(i,j)/\sigma(i,j)$. It then repeats the following step so long as the list has at least two elements.

Let j_1 and j_2 be the indices of the first two outputs on the list. Increase $x(i,j_1)$ until one of the following conditions is satisfied.

1. $x(i,+) = ST$,
2. $x(i,j_1) = \sigma(i,j_1)ST$,
3. $x(i,j_1) = B(i,j_1)$ or
4. $B(j_1) + x(i, j_1)/\sigma(i,j_1) = B(i,j_2)+ \Delta + x(i,j_2)/\sigma(i,j_2)$

If condition 1 occurs, the algorithm terminates. If either of conditions 2 or 3 occurs, remove j_1 from the list. Otherwise, move j_1 down the list so as to maintain the ordering criterion.

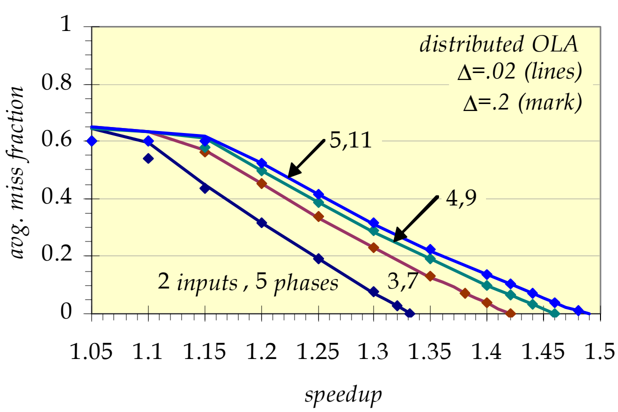

Figure 9. Miss fraction for distributed OLA on a variety of stress tests

When the list has been reduced to a single output j, the algorithm increases $x(i,j)$ until $x(i,j) = \min \{\sigma(i,j)ST, B(i, j)\}$ or until $x(i,+) = ST$, whichever occurs first.

The number of steps performed by the algorithm is at most nST/Δ. It can be implemented to run in $O((ST/\Delta)n^2)$ time, using a naive list implementation or $O((ST/\Delta)n \log n)$, if the list is replaced with a heap. Using a hardware implementation of a sorted list, this can be improved to $O((ST/\Delta)n)$ at the cost of n registers and associated comparison logic.

Figure 8 shows how distributed OLA performs on a sample stress test. This example uses a value of Δ=0.1. Comparing this to Figure 3, we see that distributed OLA reduces the miss fraction during the critical period of the last phase by about 20% relative to DBL. For this situation, distributed OLA delivers nearly ideal performance, distributing the misses evenly among the different outputs experiencing misses. Figure 9 shows how distributed OLA performs on a large number of different stress tests. Comparing these results to Figure 4, we see that distributed OLA provides the largest improvement for very small speedups. The speedups needed to reduce the misses to zero are the same for both DBL and distributed OLA.

6. Practical Considerations

While the main focus of this paper has been on establishing the theoretical foundation for robust distributed scheduling, we believe that the results are of direct practical value. First, it's important to discuss the significance of the idealized assumptions made to facilitate the analysis; specifically, the assumption that the system operation is structured in discrete phases (arrival, transfer and departure). While systems could certainly be built that adhere to this assumption, this would imply a period during which data forwarding was suspended, while scheduling was being performed. Pipelining can be used to eliminate this inefficiency. During each update period, a pipelined implementation would perform the scheduling needed to handle traffic received up to the start of the current update period. This traffic would then be allowed to proceed to the outputs during the next update period. This implies that all cells would experience a delay of between one and two update periods. While our analysis can be applied directly to systems that operate in this way, we need to relax the definition of work-conservation to reflect this delay. We say that such a system with an update period of T is T-work-conserving, if an output link is never allowed to be idle, so long as there are no cells that arrived at least $2T$ time units earlier. (Note that by this definition, crossbar schedulers that pipeline scheduling with data transfer are 1-work-conserving.)

In practice, it may be preferable not to adhere to a strict pipelining discipline, but to allow scheduling to proceed on a more or less continuous basis, with ports periodically sending their status information and *asynchronously* updating the forwarding rates of their VOQs in response to the data received from other inputs. This eliminates delays that are artificially imposed by the scheduling algorithm. Delays will still occur when the rate at which traffic arriving at an input for a given output increases suddenly, but during periods of relative rate stability there would be no unnecessary delays. It should be noted however, that while the results given here provide strong evidence that such systems can be work-conserving, they do not specifically apply to them. It would be interesting to see if one could formalize such asynchronously scheduled systems so as to enable rigorous statements about work-conservation.

Another important practical issue for distributed scheduling is the overhead of the message exchanges required by the scheduling algorithms. The practical variants of the distributed scheduling algorithms described here require that each port send and receive $2n$ values, each update period (where n is the number of ports). Using a compact floating point representation, these can be encoded with sufficient accuracy in $4n$ bytes. If the update period is chosen so that the amount of data a port can send to or receive from the interconnection network per update period is much larger than $4n$, the overhead required to communicate these values can be kept acceptably small. For a system with n=1,000 an update period of 50 µs is enough to keep the overhead below 5%.

A related issue is the computational overhead of distributed scheduling. Since the update period is necessarily a constant multiple of the number of ports, there is time to perform even moderately complex algorithms. For a system with n=1000 and a clock frequency of 200 MHz, the DBL algorithm can be executed at each port in 5 µs, a small fraction of the required update period. While more complex algorithms such as the distributed OLA algorithm are more challenging to implement in the required time, even these are within the scope of practical implementation if Δ is at least, say $ST/10$.

In this paper, we have not addressed the interconnection network itself, and how it might interact with a distributed scheduler. The performance of multistage interconnection networks with buffered switch elements has been studied in great detail, using both analysis and simulation (representative examples of analytical studies of such systems can be found in references [3,12]). The general conclusion of these studies is that these systems can provide excellent performance when carrying traffic that does not cause sustained overloads on any output links. The use of distributed scheduling can ensure that this condition is met, allowing one to consider interconnection network performance, as a largely independent issue. Most performance studies of these networks have been done assuming switch elements chips that provide buffering for just a small number of cells per port (the typical range is 2-16) and these systems are capable of throughputs exceeding 90% for switch element buffer sizes of eight or more per port. Modern ICs allow the construction of switch elements with over four thousand cells, allowing system throughputs to approach 100%. With current technology, a three stage, multi-plane, Clos-type network using dynamic routing requires roughly n switch element ICs to support n OC-192 links (for values of n ranging from about 100 to several thousand). Such a network can buffer several thousand cells per external link, allowing it to effectively smooth out any rate variations that may occur within an update period. Since rate-controlled VOQs feed traffic to the network in a smooth, rather than a bursty fashion, the magnitude of such variations can be expected to be quite limited, allowing the network to deliver cells to the outputs with only very modest queueing delays.

7. Concluding Remarks

We believe that system architectures that combine distributed scheduling and buffered, multistage interconnection networks are among the most scalable and cost-effective architectures for implementing high performance routers and switches. These architectures make it feasible today to build systems with aggregate capacities from 1 to 100 Tb/s. Continued improvements in Moore's Law will allow them to continue to scale in both line speed and aggregate system capacity. The one drawback that such systems have suffered from is that their performance can degenerate when they are subjected to the extreme traffic situations that can occur in Internet routers. While various ad-hoc flow control techniques have been used to address this issue, it has not been possible up to this point, to make rigorous statements about the performance of such systems under extreme traffic. The theoretical results developed here show that the performance of these systems can be directly comparable to the performance of unbuffered crossbars, controlled by centralized schedulers. While in both system contexts, the scheduling algorithms with the strongest theoretical guarantees are not practical to implement, these algorithms provide the insight needed to design practical variants capable of similar performance.

There are some interesting ways that this work could be extended. First, it seems entirely possible that algorithms like

DBL and distributed OLA are work-conserving for small speedups. However, proving such results seems to require either extensions to the proof techniques used here (adapted largely from earlier work on crossbar scheduling), or entirely new techniques. Establishing such a result would be of great interest from both a theoretical and a practical perspective.

Reference [11] describes distributed scheduling algorithms that support weighted-fair queueing and algorithms that seek to guarantee that packets that arrive at the same time for the same output link are forwarded at approximately the same time on that output link. The results developed here can likely be extended to allow rigorous statements about the performance of these or similar distributed schedulers.

Finally, as noted in the introduction, whereas crossbar schedulers must match inputs to outputs in a one-to-one fashion, distributed schedulers can divide the bandwidth at inputs and outputs arbitrarily. It seems likely that this difference may allow the construction of distributed schedulers with speedups significantly smaller than 2. Our failure to prove such a result may be just a consequence of our reliance on proof methods adapted from crossbar scheduling. Our simulation studies suggest that speedups close to 1.5 may be sufficient for work-conservation in distributed schedulers and we have some (so far inconclusive) analytical evidence that suggests work-conservation could be achievable for speedups of slightly less than 1.6. The establishment of such a result would be of considerable practical value and would also be interesting from a purely analytical standpoint, as it would likely require different proof techniques than those that have been employed so far.

REFERENCES

[1] Ahuja, Ravindra, Thomas Magnanti and James Orlin. *Network Flows, Theory, Applications and Algorithms*. Prentice-Hall, 1993.

[2] Anderson, T., S. Owicki., J. Saxe and C. Thacker. "High speed switch scheduling for local area networks," *ACM Trans. on Computer Systems*, 11/93.

[3] Bianchi, Giuseppe and Jonathan Turner. "Improved Queueing Analysis of Shared Buffer Switching Networks," *IEEE/ACM Transactions on Networking*, 8/93.

[4] Chang, C. S., D.S. Lee and Y.S. Jou, "Load balanced Birkhoff-von Neumann switches, Part I: one-stage buffering". Computer Communications Vol. 25. pp. 611-622, 2002.

[5] Chuang, Shang-Tse Ashish Goel, Nick McKeown, Balaji Prabhakar "Matching output queueing with a combined input output queued switch," *IEEE Journal on Selected Areas in Communications*, Dec. 1999, pp. 1030-1039.

[6] Isaac Keslassy, Shang-Tse Chuang, Kyoungsik Yu, David Miller, Mark Horowitz, Olav Solgaard, Nick McKeown, "Scaling Internet routers using optics". ACM SIGCOMM 2003, Karlsruhe, Germany, Sep. 2003.

[7] Krishna, P., N. Patel, A. Charny and R. Simcoe. "On the speedup required for work-conserving crossbar switches," *IEEE J. Selected Areas of Communications*, 6/99.

[8] McKeown, N., V. Anantharam and J. Walrand. "Achieving 100% throughput in an input-queued switch," *Proceedings of Infocom,* 1996.

[9] McKeown, N., M. Izzard., A. Mekkittikul, W. Ellersick and M. Horowitz. "The Tiny Tera: a packet switch core," *Hot Interconnects*, 1996.

[10] McKeown , Nick. "iSLIP: a scheduling algorithm for input-queued switches," *IEEE Transactions on Networking*, Vol 7, No.2, April 1999.

[11] Prashanth Pappu, Jonathan Turner and Ken Wong. "Distributed Queueing in Scalable High Performance Routers," *Proceedings of IEEE Infocom*, 4/03.

[12] Szymanski, Ted and Salman Shaikh. "Markov Chain Analysis of Packet-Switched Banyans with Arbitrary Switch Sizes, Queue Sizes, Link multiplicities and Speedups," *Proceedings of Infocom*, 4/89.

[13] Tarjan., Robert. *Data Structions and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.