Guaranteed Scheduling for Switches With Configuration Overhead

Brian Towles, *Student Member, IEEE*, and William J. Dally, *Fellow, IEEE*

***Abstract*—In this paper, we present three algorithms that provide performance guarantees for scheduling switches, such as optical switches, with configuration overhead. Each algorithm emulates an unconstrained (zero overhead) switch by accumulating a batch of configuration requests and generating a corresponding schedule for a constrained switch. Speedup is required both to cover the configuration overhead of the switch and to compensate for empty slots left by the scheduling algorithm. Scheduling algorithms are characterized by the number of configurations N_s they require to *cover* a batch of requests and the speedup required to compensate for empty slots $S_{\min}$. Initially, all switch reconfiguration is assumed to occur simultaneously. We show that a well-known exact matching algorithm, EXACT, leaves no empty slots (i.e., $S_{\min} = 1$), but requires $N_s \approx N^2$ configurations for an N-port switch leading to high configuration overhead or large batches and, hence, high delay. We present two new algorithms that reduce the number of configurations required substantially. MIN covers a batch of requests in the minimum possible number of configurations, $N_s = N$, but at the expense of many empty slots, $S_{\min} \approx 4\log_2 N$. DOUBLE strikes a balance, requiring twice as many configurations, $N_s = 2N$, while reducing the number of empty slots so that $S_{\min} = 2$. Loosening the restriction on reconfiguration times, the scheduling problem is cast as an open shop. The best known practical scheduling algorithm for open shops, list scheduling (LIST), gives the same emulation requirements as DOUBLE. Therefore, we conclude that our architecture gains no advantages from allowing arbitrary switch reconfiguration. Finally, we show that DOUBLE and LIST offer the lowest required speedup to emulate an unconstrained switch across a wide range of port count and delay.**

***Index Terms*—Optical switches, packet switching.**

NOMENCLATURE

$a_{i,j}$	Element (i, j) of matrix A.
C	Cumulative request matrix, the sum of the switch configurations requested over a period of time; the rows and columns sum to the number of configurations requested.
$C(T)$	C where rows and columns sum to T.
δ	Switching overhead in slot times.
H	Batch scheduling time in slot times (rounded up to the nearest integral number of batch times T to allow pipelining).
N	Number of switch ports.
N_s	Number of switchings per batch.
ϕ	Switch configuration interval (weight).
S	Internal speedup of switch.
P	Switch configuration/permutation matrix.
T	Batch size in slot times.

Manuscript received April 15, 2002; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor A. Orda. This work was supported by a National Science Foundation Graduate Fellowship with supplement from Stanford University and under the MARCO Interconnect Focus Research Center.

The authors are with the Computer Systems Laboratory, Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA (e-mail: btowles@cva.stanford.edu; billd@cva.stanford.edu).

Digital Object Identifier 10.1109/TNET.2003.818190

I. INTRODUCTION

OPTICAL switches based on MEMs mirrors, tunable elements, bubble switches, and similar technologies [1]–[5] have been developed to meet the exponentially increasing demand for switch bandwidth and port count. These optical switching technologies offer high bandwidth in an economical manner. Switches built with these technologies, however, require significant time to reconfigure due to mechanical settling, synchronization, and other factors. These configuration overheads range from milliseconds for bubble and free-space MEMs switches [2], [3], to 10 μs for MEMs waveguide switches [4], and as little as 10 ns for electroholographic techniques [5]. With typical cell sizes on the order of 50 ns (64 bytes at 10 Gb/s), these switches take from 0.2 to 20 000 cell times to reconfigure. Efficiently scheduling such optical switches requires algorithms that take this configuration overhead into account and optimize the resulting schedule.

Algorithms and architectures for unconstrained (zero overhead) switches often rely on the fact that switches are *stateless*: any configuration can be presented each *slot time* with no difference in switch behavior. The configuration overhead of optical switches introduces state: a switching overhead is experienced if the current switch configuration differs from the previous slot's configuration.

This paper develops an architecture and algorithms for using a constrained switch to exactly emulate the behavior of a unconstrained switch with a fixed delay. As long as the system employing the switch can tolerate the fixed delay, the emulation architecture can directly replace an unconstrained switch. In essence, emulation decouples the constraints of nonzero switching overhead from the classic switch scheduling problem. This allows designers to use optical signaling and switching directly with existing architectures and scheduling algorithms. Unlike previous algorithms that perform best effort scheduling of constrained switches [6]–[8], the algorithms we present give guaranteed performance.

The emulation architecture operates by accumulating a batch of T switch requests and then mapping this batch onto a set of $N_s < T$ switch configurations. Reducing the number of configurations reduces the time spent reconfiguring the switches and, hence, reduces the delay required for emulation. However, there is a tradeoff as aggressive reduction in the number of configurations can lead to a large number of empty slots and, hence, require a large speedup.

We first examine switching technologies that do not allow reconfiguration of some switching elements while others continue to transmit. Under this consideration, we explore three algorithms that span the design space of the number of configurations versus the number of empty slots. At one end of the design space, a well-known exact decomposition algorithm, EXACT [7], generates a schedule with no empty slots but requires $N_s \approx N^2$ configurations (where N is the number of ports) and, therefore, a very high delay. At the other extreme, we introduce a new algorithm, MIN, that generates a minimum number of configurations $N_s = N$, but leaves most slots empty and requires a switch speedup of $\Theta(\log N)$. We balance delay and speedup with another new algorithm, DOUBLE, that requires twice the minimum number of configurations $N_s = 2N$, but leaves at most half of the slots empty, thus, requiring a switch speedup of 2.

The restriction on switching times is then removed and we show that the resulting system can be considered as an open shop scheduling problem. List scheduling (LIST) [9] is then applied to the problem, ultimately yielding the same balance in switch configurations as DOUBLE. Given that LIST is a practical algorithm with the best known bound for open shop scheduling [10], [11], it follows that our architecture does not gain an advantage from a switching technology that allows some switching elements to be reconfigured while others continue to transmit.

We then compare the speedup and delay overheads of all the algorithms across the space of switch size N and delay T. Our results show that DOUBLE and LIST offer the lowest overheads of the algorithms across a wide portion of this space. EXACT offers better performance only for low port count or high delay, and MIN offers better performance only for very low delays. Viewed another way, for a fixed overhead, DOUBLE and LIST require much lower delay for emulation than EXACT at the expense of a speedup of two. For example, for a $N = 128$ port MEMS switch with a configuration time of 10 μs, EXACT requires a minimum delay of 320 ms while DOUBLE and LIST can operate with a delay of 5 ms. We also simulate the average case performance of the algorithms, which is an important design consideration in systems that include a mix of both best effort and guaranteed data for the switch. The simulations show the average number of empty slots is only a fraction of their worst case bounds, while the average number of switch configurations required is generally close to the worst case bound.

The remainder of this paper explores the design of algorithms that provide service guarantees for switches with configuration overhead in more detail. Section II introduces a simple switch model used throughout the paper. The emulation architecture is detailed in Section III. Section IV introduces three algorithms under the constraint of simultaneous reconfiguration and discusses their performance guarantees. The constraints on switching times are removed in Section V and the scheduling problem is cast as an open shop. Section VI compares all of the algorithms in terms of overhead and delay as a function of switch ports. Related work is discussed in Section VII. Finally, conclusions are drawn in Section VIII. Correctness proofs for the new algorithms are included in the Appendix.

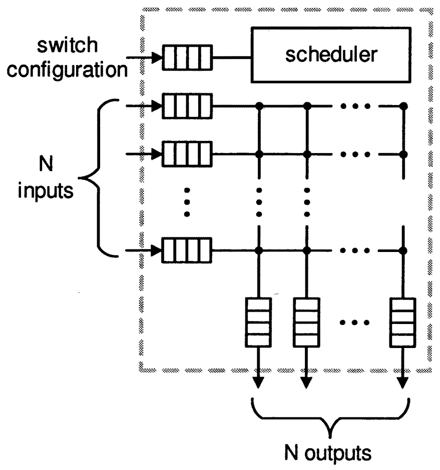


Fig. 1. Emulation architecture. The dashed boundary operates as an unconstrained switch plus a fixed delay, while the internal crossbar is implemented with a constrained switch. Both input and output buffers as well as a central scheduler are required for emulation.

II. Preliminaries

This paper deals with scheduling of a crossbar switch that can realize any one-to-one (unicast) mapping of inputs to outputs. Such a mapping is described by a *switch configuration* P, where P is a permutation matrix; when an element $p_{i,j}$ is one, input i is connected to output j for that configuration. Multicast traffic is not considered. Time is slotted and a new configuration may be provided to the crossbar each *slot time*.

Unlike typical electronic switches, the model also associates a fixed, nonzero *switching overhead* δ with each *switching event* (any change in the switch configuration). The fixed switching overhead is intended to capture all effects, such as mechanical settling times and synchronization overhead, that temporarily prevent transmission as a switching element is reconfigured. An *unconstrained* switch has $\delta = 0$, whereas a *constrained* switch has $\delta > 0$. We express δ in units of slot times.

III. Architecture

We emulate an unconstrained switch using a constrained crossbar with input and output queues (Fig. 1) where the constrained crossbar has *speedup* S to compensate for its switching overhead δ. The dashed boundary represents the standard unconstrained interface: N inputs, N outputs, and a configuration input. The speedup S refers to the ratio of the internal line rate to the input line rate. The input and output queues enable this rate mismatch by physically decoupling the internal and external lines.

A. Emulation Approach

The scheduler in Fig. 1 performs pipelined batch scheduling in four phases. In the first phase, a batch is created by accumulating the requested configurations $P(t)$ over an interval T such that

$$C(T) = \sum_{t=n}^{n+T-1} P(t).$$

Later phases may reorder the data, so incoming data is tagged with its arrival time, allowing the original order to be restored.

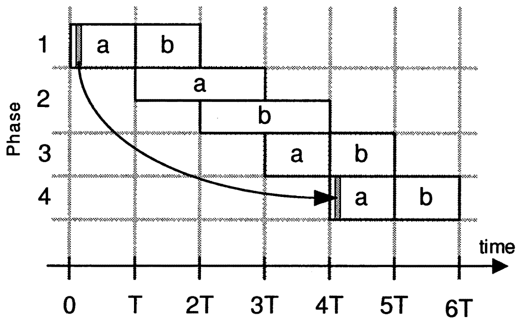


Fig. 2. Batch timeline for $H = 2T$ showing two batches, a and b, as they traverse the emulation phases. Note that both a and b can exist in the scheduling phase (phase 2) simultaneously because there are no dependencies between batches and scheduling resources can simply be duplicated. A single packet is also shown in gray with an arc connecting its entrance and exit times from the emulation architecture.

The second phase computes a switch schedule for the batch. While the time to compute a batch's schedule H is assumed to be a multiple of the batch time T, it is not necessary that $H = T$. The pipeline diagram shown in Fig. 2 illustrates the case when $H = 2T$. Since each batch processed by the switch is independent, multiple batches may be scheduled in parallel given that H/T sets of switch scheduling hardware exist.

Once the schedule for a batch is computed, it is executed by the constrained switch during the third phase. The critical guarantee necessary for the architecture to emulate an unconstrained switch is that the third phase *never* takes longer than the batch time T. This guarantee also ensures that no data element stays in the input queues for more than $2T + H$ slot times. After traversal, data is reordered as it is stored in the output buffers.[1]

Finally, the fourth phase simply sends the data from the output buffers onto the output lines in the same order it entered the switch. As shown by the arc between an arriving data element and its departure from the switch (Fig. 2), this relationship implies a delay bound of $2T + H$ when H slot times are reserved for the second phase scheduling. Therefore, the outputs exactly emulate the behavior of a corresponding unconstrained switch plus the fixed delay of $2T + H$.

As expected, the amount of storage required in the architecture grows linearly with T. Let L indicate the number of bits sent to a single input port during a slot time. Considering one port, a batch is held for $2T + H$ slot times in the input buffers and since a new batch is started each T slot times, enough buffers for $(2T + H)L$ bits of data are required in the input stage. Similarly, data is held for $2T$ slot times in the output stage, requiring $2TL$ bits of buffering. So, considering all ports, the architecture needs $(4T + H)LN$ bits of total buffering.

B. Emulation Requirements

To compensate for the overhead of switch configuration and slots left empty by the scheduling algorithm, the emulation architecture must operate with a speedup S that depends on the batch size T as illustrated in Fig. 3. S is selected to ensure that $C(T)$ can be completely transmitted during the third phase of

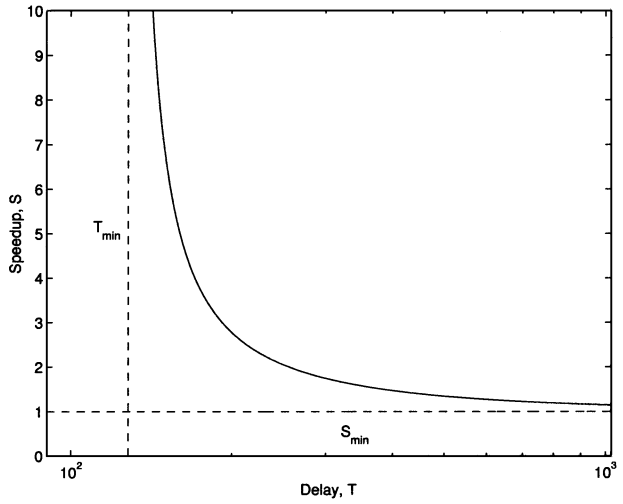


Fig. 3. Speedup required for emulation ($N_s = 128$, $\delta = 1$). Both $T_{\min}$ and $S_{\min}$ are assumed to be constants for this example.

the emulation algorithm. The time spent on configuration overhead during each batch of N_s configurations is $T_{\min} = \delta N_s$, the left asymptote of the S versus T curve. This leaves time $T - T_{\min}$ to send T slots of data. If the scheduling algorithm exactly filled each of the slots with data, the speedup required would be $S_{\text{exact}} = T/(T - T_{\min})$.[2]

Not all algorithms completely fill the slots, however. So the total number of slots used by a scheduling algorithm T_s can be greater than T in general. Thus, the speedup required to compensate solely for these empty slots is $S_{\min} = T_s/T$, which gives the bottom asymptote of the S versus T curve. Viewed another way, the fraction of slots filled by the scheduling algorithm is $1/S_{\min}$. So, for example, if half the slots are filled with data, an additional speedup of $S_{\min} = 2$ is required beyond the speedup S_{exact} necessary to compensate for switching overhead.

Multiplying these two speedups gives the total speedup required for a particular batch size T

$$S = \frac{S_{\min}T}{T - T_{\min}} = \frac{S_{\min}T}{T - \delta N_s}, \qquad T > T_{\min}.$$

This relationship can be also rewritten to give the batch size required for a particular speedup S

$$T = \frac{ST_{\min}}{S - S_{\min}} = \frac{\delta N_s S}{S - S_{\min}}, \qquad S > S_{\min}.$$

IV. Scheduling With Simultaneous Reconfigurations

The scheduling task is a time-slot assignment problem. Given an input–output request matrix C, assign a switch traversal time for each element in C so that the total transmission time is minimized. Emulation also requires *guarantees* about the performance of scheduling algorithms. That is, for any matrix C and switching overhead δ, the worst case transmission time required for a scheduling algorithm must be bounded.

We first approach this problem by finding decompositions of the request matrix C into N_s permutation matrices, such that $N_s < T$. Specifically, a set of switch configurations

[1]If a particular design only requires that packets between each input–output pair remain in order, no ordering tags are required and no reordering is required at the end of the third phase.

[2]While both $T_{\min}$ and $S_{\min}$ are constants in all the algorithms presented in this paper, it also possible for their values to vary with S, T, and/or δ.

$P(1),\ldots,P(N_s)$ and corresponding weights $\phi(1),\ldots,\phi(N_s)$ that *covers* C is found during phase 2 of the pipeline:

$$\sum_{k=1}^{N_s}\phi(k)p_{i,j}(k)\geq c_{i,j}, \qquad \forall i,j\in\{1,\ldots,N\}.$$

In the case of equality for all i and j, the switch configurations *exactly cover* C. Then, during phase 3, the constrained switch is configured in the sequence $P(1),\ldots,P(N_s)$ with each configuration held for $\phi(1),\ldots,\phi(N_s)$ slot times. All of the switching elements in the constrained switch are reconfigured simultaneously.

The requirement of simultaneous reconfiguration arises due to technological constraints of specific systems. For example, issues such as optical crosstalk in a free-space optical switch or coupling between mechanical switching elements may prevent switches from being reconfigured while others continue to transmit. Also, related problems, such as scheduling SS/TDMA systems (Section VII), require simultaneous reconfiguration. This constraint is loosened in Section V.

This section presents several algorithms for achieving guaranteed performance with simultaneous reconfigurations and examines the tradeoff between the number of switch configurations used to cover the matrix and the number of empty slots left by the algorithm. An example of this tradeoff is illustrated in Fig. 4. First, a request matrix C is decomposed into four switch configurations that exactly cover C [Fig. 4(a)]. The accompanying time-slot assignment diagram shows the connection of inputs (shown vertically) to particular outputs, denoted by slot labels. The shaded segments show the switching time required between different configurations. An alternative decomposition of C gives only three switchings, but the corresponding time-slot assignment contains empty slot times [Fig. 4(b)]. Since each configuration is held for the maximum time of all the elements contained within it, some slots are left unused. From this simple example, it should be clear that fewer switchings require less overhead time, but at the potential cost of leaving slots empty during switch traversal. This tradeoff is quantified in the following sections.

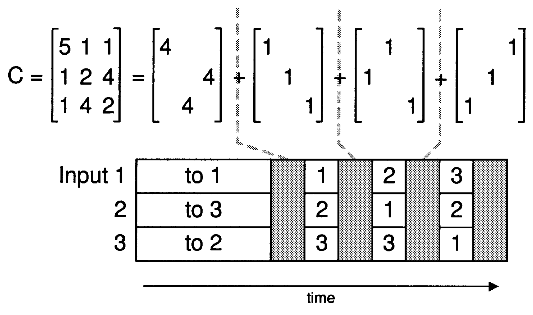
A. Exact Covering

A well-known decomposition of any matrix $C(T)$ [7], [12] exactly covers the matrix in at most $N_s=N^2-2N+2$ switch configurations.

Theorem 1: $N_s=N^2-2N+2$ switch configurations and positive integer weights $\phi(1),\ldots,\phi(N_s)$ are necessary and sufficient to exactly cover any $N\times N$ matrix $C(T)$.

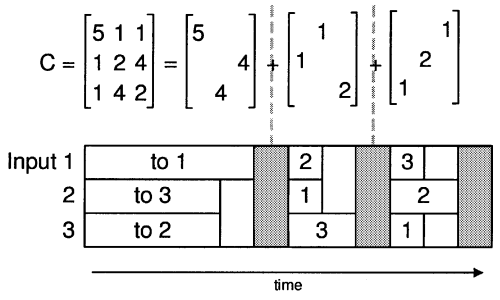
Proof: As noted in [13, p. 36], necessity is proved in [14] and sufficiency in [12]. □

Several algorithms are suggested in [7] to realize the lower bound on the number of configurations required. These algorithms include optimizations to improve the average number of configurations, but we consider a simple algorithm that only meets the bound of $N_s=N^2-2N+2$ configurations (Algorithm 1). The EXACT algorithm repeatedly performs maximum-size matchings on the nonzero elements of C (Step 2). The weight of the corresponding configuration is taken as the minimum value of all the elements of C included in the match (Step 3). This ensures that at least one element of the request matrix is zeroed per iteration. Then the configuration is subtracted from request matrix and this procedure is repeated until all entries of C have been zeroed (Step 4). This algorithm always terminates in N^2-2N+2 iterations, meeting the lower bound. Since each step requires a maximum-size matching of complexity $O(N^{2.5})$, the overall run time of EXACT is $O(N^{4.5})$.



C = [5 1 1; 1 2 4; 1 4 2] = [4 ; 4; 4] + [1; 1; 1] + [1; 1; 1] + [1; 1; 1]
Input 1
2
3
to 1
to 3
to 2
1
2
3
2
1
3
3
2
1
time

(a)



C = [5 1 1; 1 2 4; 1 4 2] = [5; 4; 4] + [1; 1 2] + [1; 2; 1]
Input 1
2
3
to 1
to 3
to 2
2
1
3
3
2
1
time

(b)

Fig. 4. Tradeoff between fewer switch configurations and empty slots ($\delta=1$). (a) Decomposition into four configurations with no unused slots. (b) Decomposition into three configurations resulting in the introduction of empty slots into the schedule.

```
Algorithm 1 Exact Covering (EXACT)
Step 1) Initialization. Set $i \leftarrow 1$ and $A \leftarrow C(T)$.
Step 2) Bipartite match. Construct a bipartite
        graph from $A$ where each nonzero entry of
        $A$ has a corresponding edge in the graph.
        Find a maximum-size matching $M$ of this
        graph.
Step 3) Schedule. Construct a permutation $P(i)$
        which corresponds to the matching $M$.
        Set the weight based on the minimum
        entry value of $A$ corresponding to the
        edges of $M$ : $\phi(i) \leftarrow \min_{(e,f)\in M} a_{e,f}$.
Step 4) Update and loop. Set $A \leftarrow A - \phi(i)P(i)$ and
        $i \leftarrow i+1$. If any nonzero entries of $A$ re-
        main, go to Step 2. Otherwise end.
```

Given the bound on the number of switch configurations, the total amount of switching overhead can be determined and, therefore, the required speedup can be calculated.

Corollary 1: A speedup of

$$\frac{T}{T-\delta(N^2-2N+2)}$$

is sufficient to schedule $C(T)$ in T slot times.

Proof: This follows directly from the number of switchings $N_s=N^2-2N+2$ and the minimum speedup of $S_{\min}=1$ required for an exact covering. □

Note that the minimum fixed delay experienced by the switch, $T_{\min} = \delta(N^2 - 2N + 1)$, must grow at least with the square of the number of ports on the switch. This implies the amount of storage must also grow with $O(N^2)$. In a system where the bandwidth between input–output pairs is expensive relative to the cost of providing the storage and tolerating the fixed delay, exact covering is an attractive approach. Since N^2-2N+1 configurations are necessary to exactly cover C, further reducing the number of switchings would introduce empty slots leading to a waste of the expensive bandwidth. Alternatively, in systems with inexpensive bandwidth a designer may be willing to trade that bandwidth for a smaller fixed delay and less storage requirements.

B. Minimum Switchings

While $O(N^2)$ configurations are necessary to exactly cover C, it is possible to cover any C with as few as N configurations. This is clearly the minimum number of configurations as C has N^2 nonzero entries in general and each configuration covers at most N of these entries. However, the use of fewer configurations introduces empty slots which must be overcome with speedup. In this section, we show that cost of these empty slots can be quite significant: for $N_s = N$, $S_{\min}$ is $\Theta(\log N)$.

Theorem 2: To transmit a general cumulative schedule matrix $C(T)$ in N switch configurations, $S_{\min}$ must be at least $\Omega(\log N)$ for $T > N$.

Proof: An adversarial matrix C is constructed by the following algorithm.[3]

```
Step 1) Initialization. Create two $N \times N$
      matrices, $A$ and $B$. Initialize all
      entries of $A$ to zero and all en-
      tries of $B$ to one. Set $i \leftarrow 1$ and
      $j \leftarrow 1$.
Step 2) Build $A$. Fill the submatrix
      $$A(i : i+j-1, i : i+j-1) = \frac{T'}{j}$$
      where $T' = T - N$. Set $i \leftarrow i + j$ and
      $j \leftarrow j + 1$. If $i + j > N$ go to Step 3,
      otherwise repeat Step 2.
Step 3) Create $C$. Set $C \leftarrow A + B$ and $I \leftarrow i$.
```

From the construction of A (Fig. 5), it is clear that each row and column sums to T'. The rows and columns of B each sum to N and, therefore, each row and column sum of C is T.

The addition of the B matrix to A guarantees that there are no nonzero elements in C. Since all N^2 elements are covered in N switch configurations, each configuration must cover N unique elements. This implies that each element is included in exactly one switch configuration. For any scheduling algorithm that covers C, a switch configuration $P(1)$ will contain the $T'+1$ entry (element $c_{1,1}$). Two switch configurations are required to cover all the $T'/2+1$ entries, so at least one of the entries will be in a configuration $P(2)$, where $P(2) \neq P(1)$. Likewise, one $T'/3+1$ entry will be in $P(3)$, where $P(3) \neq P(2)$ and

[3]For clarity, this proof assumes that all parameters are such that the elements of C are integers. However, the same result holds if the elements of the constructed C are all rounded down to the nearest integer.

$$A = \begin{bmatrix} T' & & & & & & \\ & T'/2 & T'/2 & & & & \\ & T'/2 & T'/2 & & & & \\ & & & T'/3 & T'/3 & T'/3 & \\ & & & T'/3 & T'/3 & T'/3 & \\ & & & T'/3 & T'/3 & T'/3 & \\ & & & & & & \ddots \end{bmatrix}$$

Fig. 5. Portion of adversarial matrix which requires $\Omega(T \log N)$ empty slots to be scheduled in a minimum number of configurations.

$P(3) \neq P(1)$. This argument continues for I of the switch configurations. Since the time required for a switch configuration is the maximum of all elements in that configuration, switching $P(1)$ through $P(I)$ requires at least

$$(T'+1) + \left(\frac{T'}{2}+1\right) + \ldots + \left(\frac{T'}{I}+1\right) > T' \ln I.$$

From the above algorithm, I is the largest integer such that $\sum_{i=1}^{I} i \leq N$, or

$$I = \left\lfloor \frac{\left(\sqrt{1+8N}-1\right)}{2} \right\rfloor > \sqrt{2N} - \frac{3}{2}.$$

Substituting yields the total number of time slots required,

$$(T-N)\ln(I) > (T-N)\ln\left(\sqrt{2N} - \frac{3}{2}\right).$$

Therefore, an $S_{\min}$ of at least $\Omega(\log N)$ is required. □

This result shows that regardless of the algorithm used, scheduling C so that there are only N switch configurations requires $S_{\min} = \Omega(\log N)$ in general. A simple algorithm MIN (Algorithm 2) shows this bound on the minimum speedup is also sufficient.[4] The algorithm's running time is dominated by N maximum size matchings, for a total time complexity of $O(N^{3.5})$.

```
**Algorithm 2** Minimum switchings (MIN)
Step 1) Initialization. Create an $N \times N$ indi-
      cator matrix $B$ with all entries set to
      one. Set $d \leftarrow 2$ and $k \leftarrow 1$.
Step 2) Identify large elements. Define the
      $N \times N$ matrix $A$ such that
      $$a_{i,j} = \begin{cases} 1 & \text{if } c_{i,j} > \frac{T}{d} \text{ and } b_{i,j} = 1 \\ 0 & \text{otherwise} \end{cases}$$
Step 3) Color. Construct the bipartite graph
      $G_A$ from A (zero entries do not have a
      corresponding edge). Perform a minimal
      edge coloring of $G_A$.
Step 4) Schedule. Set $c \leftarrow 1$.
  Step 4a. Partition edges. Let the matching
         $M_c$ be the subset of edges in $G_A$
         assigned to color $c$. Take any
         subset of edges $E_a \subseteq M_c$, such that
         $|E_a| = \lceil |M_c|/2 \rceil$. Then $E_b \leftarrow M_c - E_a$.
  Step 4b. Schedule $E_a$. Construct the bipar-
         tite graph $G_B = (E_B, V_B)$ from $B$. Re-
         move edges from $E_a$ which have been
         previously scheduled by setting
         $E_a \leftarrow E_a \cap E_B$. Then, for each edge in
         $E_a$, remove the corresponding edge,
```

[4]The algorithm and analysis presented assume $N \geq 8$ for simplicity. Cases where $N < 8$ can be handled by slightly modifying Steps 4–5.

```
                that edge's endpoints, and edges
                incident to those endpoints from
                $G_B$. Find the maximum-size matching
                $M_B$ on the remaining vertices and
                edges of $G_B$. Construct the config-
                uration $P(i)$ from the combination
                of the two matchings $M_c \cup M_B$ and
                set the weight $\phi(i) \leftarrow \lfloor 2T/d \rfloor$. Set
                $B \leftarrow B - P(i)$ and $i \leftarrow i+1$.
     Step 4c. Schedule $E_b$. Repeat the procedure
                of Step 4b, but for the edges of $E_b$
                instead of $E_a$.
     Step 4d. Loop over colors. Set $c \leftarrow c+1$. If
                $c \leq d-1$, then go to Step 4a. Other-
                wise continue to Step 5.
Step 5. Loop. Set $d \leftarrow 2d$. If $(i-1)+2(d-1) \leq N/4$,
      then go to Step 2. Otherwise continue to
      Step 6.
Step 6. Finish. Construct the bipartite graph
      $G_B$ from $B$. Perform a maximum-size
      matching on $G_B$ and produce the switch
      schedule $P(i)$. Set $\phi(i) \leftarrow \lfloor 2T/d \rfloor$, $B \leftarrow$
      $B - P(i)$, and $i \leftarrow i+1$. Repeat Step 6 until
      there are no nonzero elements remaining
      in $B$.
```

The MIN algorithm generates a logarithmic bound on the total configuration weight and, therefore, the number of empty slots, by first identifying the largest unscheduled elements of C at the beginning of the outer loop (Steps 2–5). Large elements are defined as being greater threshold value T/d, which is halved during each iteration of the outer loop. Steps 3–4 ensure that the elements greater than a particular threshold can always be scheduled in roughly $2d$ configurations. Since previous iterations guaranteed that all elements greater than $2T/d$ were scheduled, the total weight produced by each outer loop iteration is roughly $(2T/d)(2d) = 4T$, which is constant in N.

Only approximately one quarter of the elements of C are scheduled in this outer loop, and since the number of configurations produced by the outer loop doubles per iteration, this gives a total weight of approximately $4T \log_2 N$. The remaining $3N/4$ configurations are created in Step 6 with a weight of roughly $16T/N$ each, giving a total weight of $(3N/4)(16T/N) = 12T$, which does not affect the overall logarithmic behavior of the algorithm.

To guarantee these bounds on the total configuration weight, Steps 3–4 must schedule all the elements greater than the threshold of T/d in approximately $2d$ configurations. Step 2 finds the large, unscheduled elements of C in the matrix A. Because the rows (columns) of C sum to T, there can be at most $d-1$ elements greater than or equal to this threshold in each row (column) of A. This allows Step 3 to perform an edge-coloring of the corresponding bipartite graph G_A in at most $d-1$ colors due to the classical result of König.

For each of the $d-1$ colors, Step 4 produces two configurations, meeting the bound of $2d$ total configuration per iteration of the outer loop. Considering a single color of edges, Step 4 first divides this group of edges in half (Step 4a). Then Step 4b builds a perfect matching[5] that includes the first half of these edges and Step 4c performs the same task for the second half. Since each configuration in a minimum switchings algorithm must cover N unique entries of C, perfect matchings, which correspond to full permutations, must found at these steps. Also, if the edges are not split into two subsets, it is not always be possible to find a perfect matching that contains all the edges. However, by relaxing this constraint so that each perfect matching needs to only contain half the colored edges, such a matching provably exists as long as there are more than $3N/4$ entries left to be scheduled in C (see the Appendix). This condition on the number of unscheduled entries is ensured by Step 5 and explains the limit of $N/4$ configurations produced in Steps 2–5.

Fig. 6 shows an example execution of the MIN algorithm for a matrix with $N = 32$ and $T = 32$. For simplicity, only a portion of the matrices and the first several steps are illustrated. In the first iteration of the example, $d = 2$ and the first threshold is $T/d = 16$. All entries >16 are considered for scheduling and indicated in A. For the first iteration, A requires only $d-1 = 1$ color in Step 3. Then, during Step 4a, the nonzero entries of A are partitioned into two subsets E_a (circled) and E_b (not circled). The elements of E_a are a subset of a perfect matching found in Step 4b, which is used as schedule $P(1)$ with weight $\phi(1) = 2T/d = 32$. Similarly, the elements of E_b are scheduled in Step 4c. After both steps, B is shown with zero entries corresponding to the scheduled elements of C.

The outer loop is repeated for $d = 4$ and all unscheduled elements in C greater than $T/d = 8$ are indicated in A. Again, A is colored using $d-1 = 3$ colors. Schedules $P(3)$ and $P(4)$ correspond to the first color, while the remaining colors (shown in gray) are used for schedules $P(5)$ through $P(8)$. In Step 5, $(i-1)+2(d-1) = 8+14 = 22$ is greater than $N/4 = 8$, so the algorithm goes to Step 6 and creates the remaining schedules.

The general operation of MIN is verified in the Appendix.

Theorem 3: To cover a general cumulative schedule matrix $C(T)$ with N switch configurations $S_{\min} = 4T(4 + \log_2 N)$ is sufficient.

Proof: Let m be the number of iterations of the outer loop of MIN (Steps 2–5). m is the largest integer such that

$$2\sum_{i=1}^{m}(2^i - 1) = 2(2^{m+1} - m - 2) \leq \frac{N}{4}.$$

Using m, the total weight of the schedules produced from Steps 2–5 is then

$$2\sum_{i=1}^{m}(2^i - 1)\left\lfloor \frac{2T}{2^i} \right\rfloor < 4Tm.$$

The total weight produced during Step 6 is

$$\left(N - 2(2^{m+1} - m - 2)\right)\left\lfloor \frac{2T}{2^{m+1}} \right\rfloor \leq \frac{2T(N+2m+4)}{2^{m+1}} - 4T.$$

By conservatively estimating m as $\lfloor \log_2(N/16) \rfloor$, a bound on the total weight is then

$$4T\left\lfloor \log_2\left(\frac{N}{16}\right)\right\rfloor + \frac{2T\left(N + 2\left\lfloor \log_2\left(\frac{N}{16}\right)\right\rfloor + 4\right)}{2^{\left\lfloor \log_2\left(\frac{N}{16}\right)\right\rfloor + 1}} - 4T.$$

[5]A perfect matching is a subset of edges such that each vertex is incident with exactly one edge in that subset.

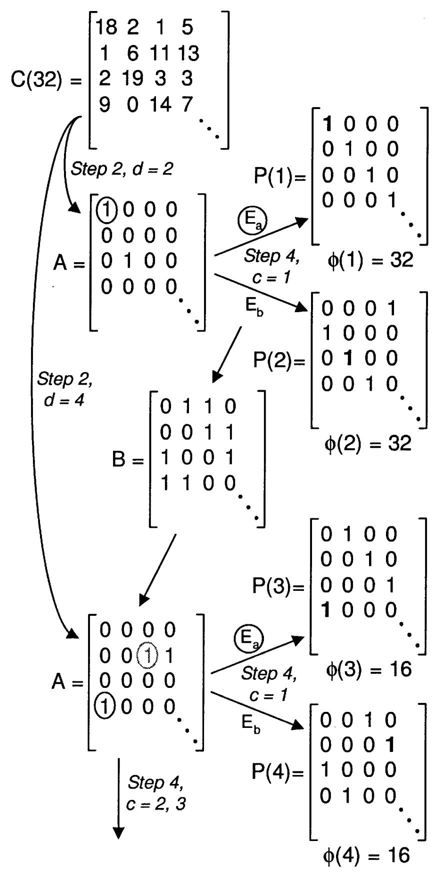

C(32) = [18 2 1 5; 1 6 11 13; 2 19 3 3; 9 0 14 7]
Step 2, d = 2
A = [1 0 0 0; 0 0 0 0; 0 1 0 0; 0 0 0 0]
E_a
Step 4, c = 1
E_b
P(1)= [1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1]
ϕ(1) = 32
P(2)= [0 0 0 1; 1 0 0 0; 0 1 0 0; 0 0 1 0]
ϕ(2) = 32
Step 2, d = 4
B = [0 1 1 0; 0 0 1 1; 1 0 0 1; 1 1 0 0]
P(3)= [0 1 0 0; 0 0 1 0; 0 0 0 1; 1 0 0 0]
ϕ(3) = 16
A = [0 0 0 0; 0 0 1 1; 0 0 0 0; 1 0 0 0]
P(4)= [0 0 1 0; 0 0 0 1; 1 0 0 0; 0 1 0 0]
ϕ(4) = 16
Step 4, c = 2, 3

Fig. 6. Example execution of MIN ($N = 32, T = 32$). Only the first few steps are shown.

Through further simplification, this expression can be bounded by $4T(4 + \log_2 N)$. Therefore, the minimum speedup is sufficient. □

Corollary 2: A speedup of

$$\frac{4T(4 + \log_2 N)}{T - \delta N}$$

is sufficient to schedule $C(T)$ in T slot times.

Proof: This follows directly from the number of switchings $N_s = N$ and the minimum speedup of $S_{\min} = 4(4+\log_2 N)$ required for MIN. □

So, while successfully reducing the number of configurations to the minimum possible, the amount of speedup required to support this few switchings grows with $\log N$. This could be an effective tradeoff for switches with inexpensive bandwidth or a small number of ports. However, for larger switches, the required speedup factor could be too expensive. In this case, a more attractive alternative may be to use a near-minimum number of configurations.

C. Near-Minimum Switchings

As described in the previous section, using the minimal number of switchings requires a speedup of at least $\log N$. In this section, we show that by allowing $2N$ switchings, the minimum speedup $S_{\min}$ can be reduced to approximately two. Most importantly, the minimum speedup is no longer a function of N. This approach has the advantage of the EXACT algorithm, a small constant speedup, combined with a number of switchings that grows linearly with N. The DOUBLE algorithm (Algorithm 3) produces schedules with these properties in $O(N^2 \log N)$ time using the edge-coloring algorithm of [15].

```
Algorithm 3 Near-minimum switchings (DOUBLE)
Step 1) Split $C$. Define an $N \times N$ matrix $A$ such
        that
            $$a_{i,j} = \left\lfloor \frac{c_{i,j}}{\frac{T}{N}} \right\rfloor.$$
Step 2) Color $A$. Construct the bipartite
        multigraph $G_A$ from $A$ (the number of
        edges between vertices is equal to the
        value of the corresponding entry of $A$).
        Find a minimal edge-coloring of $A$. Set
        $i \leftarrow 1$.
Step 3) Schedule coarse. For a specific color
        in the edge-coloring of $G_A$, construct a
        switch configuration $P(i)$ from the edges
        assigned that color. Set $\phi(i) \leftarrow \lceil T/N \rceil$ and
        $i \leftarrow i + 1$. Repeat Step 3 for the each of
        the colors in $G_A$.
Step 4) Schedule fine. Find any $N$ nonoverlap-
        ping switch schedules $P(N + 1), \ldots, P(2N)$
        and set $\phi(N+1), \ldots, \phi(2N)$ to $\lceil T/N \rceil$.
```

DOUBLE works by separating C into *coarse* and *fine* matrices and devotes N configurations to each. The algorithm first generates the coarse matrix A by dividing the elements of C by T/N and taking the floor. The rows and columns of A sum to at most N, thus the corresponding bipartite multigraph can be edge-colored in N colors. Each subset of edges assigned to a particular color forms a matching, which is weighted by $\lceil T/N \rceil$. The fine matrix for C does not need to be explicitly computed because its elements are guaranteed to be less than $\lceil T/N \rceil$. Thus, any N configurations that collectively represent every entry of C, each weighted by $\lceil T/N \rceil$, can be used to cover the fine portion.

An example execution of DOUBLE is shown in Fig. 7. The algorithm begins by creating the coarse matrix A by dividing each element in C by T/N and taking the floor. So, in the example, entry (1,1) of A contains $\lfloor 16/(T/N) \rfloor = 16/(16/4) = 4$. The resulting matrix A has row and column sums $\leq$4, ensuring that it can be edge colored with 4 colors (Step 2). Then, the edges assigned to each color are converted to schedules in Step 3. For example, $P(1)$ corresponds to the subset of edges assigned to color 1 during Step 2. Also, some of the schedules may not be complete permutations because the row and column sums of A are less than N, such as $P(3)$ and $P(4)$, but it is still guaranteed that all the elements of A are covered. In general, Step 3 creates at most N matchings with weight $\lceil T/N \rceil$, for a total weight of approximately T.

Step 4 picks four nonoverlapping schedules, $P(4)$ through $P(8)$, and each is assigned a weight of $\lceil T/N \rceil = 4$. In general, Step 4 creates the same total weight as Step 3: approximately T. Therefore, the total weight to schedule $C(T)$ using DOUBLE is approximately $2T$ and $S_{\min} = 2$. The general operation of DOUBLE is verified in the Appendix.

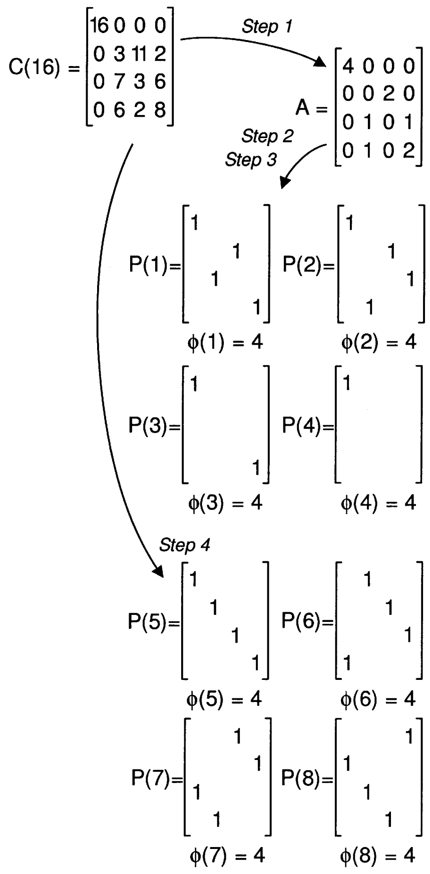


Fig. 7. Example execution of DOUBLE ($N = 4, T = 16$).

The required speedup is now simply derived from the weights assigned by DOUBLE.

Theorem 4: To transmit a general cumulative schedule matrix $C(T)$ in $2N$ switch configurations $S_{\min} = 2$ is sufficient when T is a multiple of N.

Proof: DOUBLE produces $2N$ switch configurations, each with a weight of $\lceil T/N \rceil$. Summing these weights

$$2N \left\lceil \frac{T}{N} \right\rceil = 2T.$$

Therefore, the minimum speedup is sufficient. □

Corollary 3: A speedup of

$$\frac{2T}{T - 2\delta N}$$

is sufficient to schedule $C(T)$ in T slot times when T is a multiple of N.

Proof: This follows directly from the number of switchings $N_s = 2N$ and the minimum speedup of $S_{\min} = 2$ required for DOUBLE. □

V. Scheduling With Arbitrary Reconfigurations

In the previous section, we presented several algorithms that achieve guaranteed performance with simultaneous reconfigurations. When the underlying technology allows switches to be reconfigured while others continue to transmit, a more general scheduling problem can be considered. In fact, scheduling a switch with arbitrary reconfigurations can be formulated as an *open shop scheduling* problem—a classic problem in operations research. We present this formulation and apply the list scheduling algorithm of [9] to the problem. List scheduling is shown to offer identical guarantees to the DOUBLE algorithm at a slightly lower algorithmic complexity.

A. Open Shop Formulation

An open shop is a collection of jobs J and machines M. Each individual job $j_i \in J$ is a set of tasks $j_i = \{t_{i,1}, \ldots t_{i,k}\}$, which must be executed under several constraints: each task $t_{i,o}$ must be executed on machine $m_o \in M$ for a total of $dur(t_{i,o})$ time and each machine can only execute at most task at any given time. Since the shop is "open," the tasks of each job may be executed in any order.

For a given open shop problem, we are concerned with minimizing the *makespan* or the time required to complete all the jobs. In open shop literature, this problem is often abbreviated as $O||C_{\max}$ and is known to be NP-hard for more than three machines [16].

Using this notation, the constrained switch scheduling problem can be expressed as an open shop problem. For a request matrix C, the switch speedup S, and switching overhead δ, the individual tasks of a corresponding open shop are defined as

$$dur(t_{i,j}) = \delta + \frac{c(T)_{i,j}}{S}.$$

The jobs correspond to inputs of the switch, $|J| = N$, and the machines correspond to the outputs, $|M| = N$.

Then, any constrained switch scheduling problem can be solved using algorithms developed for open shop scheduling. When a particular task $t_{i,j}$ is scheduled on machine m_j, the switch is configured to connect input i to output j. Since any valid open shop schedule runs the task $t_{i,j}$ for $dur(t_{i,j})$ time, the corresponding constrained switch schedule has enough time to configure the switch, which requires δ time, and pass the data from input i to output j, which requires $c_{i,j}/S$ time.

B. List Scheduling

List scheduling (LIST) is a greedy algorithm that can be used to approximate the optimal open shop schedule within a factor of two [9]. LIST starts by assigning a job to each machine. If multiple jobs are contending for a single machine, one of the jobs is chosen arbitrarily. Then the initial schedule of jobs to machines continues until a task is complete and the corresponding machine is freed. Once a machine is idle, any job not currently assigned to another machine that also has a task remaining for the free machine is placed on that machine. Again, contention is resolved arbitrarily. This continues until all the jobs are complete. Creating a schedule with LIST requires $O(N^2)$ time.

The best bounds on the maximum schedule length produced by LIST is the sum of the time to process the longest job (sum of its tasks' durations) and the time to process the most heavily used machine (sum of all tasks for that machine) [11]. Using the formulation for constrained switches described in the previous section, the maximum schedule length $C_{\max}$ produced by list scheduling is

$$\begin{aligned} C_{\max} &\le \max_i \sum_j dur(t_{i,j}) + \max_j \sum_i dur(t_{i,j}) \\ &\le \max_i \sum_j \left(\delta + \frac{c_{i,j}}{S}\right) + \max_j \sum_i \left(\delta + \frac{c_{i,j}}{S}\right) \\ &\le 2\left(\delta N + \frac{T}{S}\right). \end{aligned}$$

To guarantee the operation of the emulation architecture, the speedup provided must ensure the schedule length is at most T slot times, therefore, S is chosen such that $T = 2(\delta N + T/S)$. Rewriting

$$S = \frac{2T}{T - 2\delta N}.$$

Using this, we know the minimum speedup for list scheduling is $S_{\min} = 2$ and the minimum delay is $T_{\min} = 2\delta N$.

Although LIST allows less restricted assignment of switching times compared to the simultaneous switching algorithms presented in Section IV, it offers no worst case advantage over the DOUBLE algorithm. However, it does have a slight improvement in running time.

VI. Discussion

A. Analysis of Design Tradeoffs

The previous sections detailed four algorithms for unconstrained switch emulation. Given these algorithms, which is the most appropriate for a particular system? The answer depends on the relative costs of bandwidth, delay, storage, and the switching overhead δ in the system.

If the system designer is insensitive to delay and storage requirements, but considers bandwidth expensive, then the EXACT algorithm is most likely an appropriate design choice. However, exact scheduling can lead to large delays, even with feasible system parameters. For example, consider a 128-port switch with 10-Gb/s input lines and a 64-byte slot (slot time of 50 ns). Fast MEMS mirror switches are used, which have a switching time of $\delta = 200$ or 10 μs [4]. For exact matching, $T_{\min} = \delta N_s$ is approximately 3.2 million slot times or 160 ms, which makes the minimum fixed delay $2T + H$ equal to 320 ms plus the scheduling time. This delay is obviously unacceptable for many switching applications.

The minimum switching algorithm MIN greatly reduces the fixed delay over the exact algorithm, but at the cost of increased speedup. In our example 128-port switch, MIN reduces the minimum fixed delay to 2.5 ms, but requires a minimum speedup of $4(4 + \log_2 N) = 44$.

DOUBLE provides a balance between the these two extremes. For the 128-port switch, a minimum fixed delay of 5 ms and minimum speedup of 2 are necessary. So, compared to the exact algorithm, a speedup of 2 reduces the fixed delay by a factor of 128. Alternatively, DOUBLE allows a switching overhead δ that is 128 times greater than the exact algorithm for the same fixed delay. Assuming there is a cost benefit in slower switches, the potential savings from using slower switches may more than offset the cost required to provide a speedup of 2.

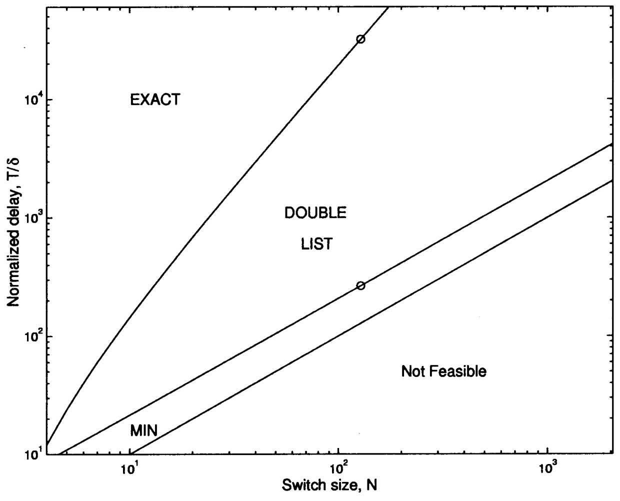
TABLE I
Summary of Algorithm Costs (Also Shown for $N = 128, \delta = 200$)

	EXACT	MIN	DOUBLE	LIST
N_s	$N^2 - 2N + 2$ (16 130)	N (128)	$2N$ (256)	$2N$ (256)
$S_{\min}$	1 (1)	$4(4 + \log_2 N)$ (44)	2 (2)	2 (2)
$T_{\min}$	$\delta(N^2 - 2N + 2)$ (3 226 000)	δN (25 600)	$2\delta N$ (51 200)	$2\delta N$ (51 200)
Requires arbitrary reconfig	*no*	*no*	*no*	*yes*

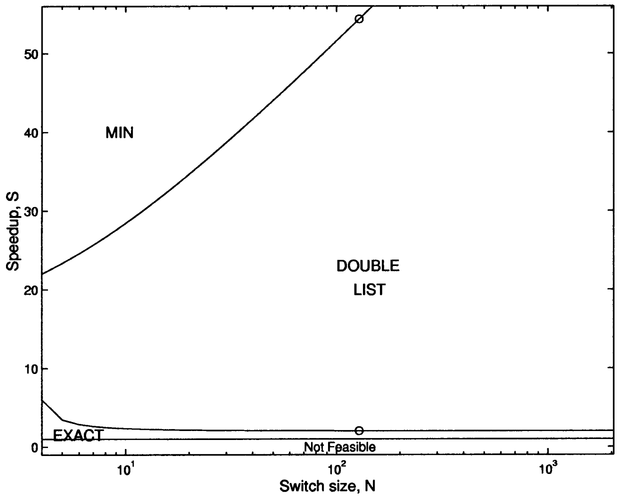
LIST requires that switches to be reconfigured at arbitrary times, in contrast to the simultaneous reconfiguration of EXACT, MIN, and DOUBLE. Despite this additional flexibility, it does not offer any improvement in its worst case guarantees. Moreover, list scheduling is the best known practical algorithm for generating open shop schedules in polynomial time [10], [11]. It remains an open question whether a practical algorithm can improve the speedup or delay requirements by taking advantage of arbitrary switch configurations.

A summary of the costs for all scheduling algorithms is shown in Table I, and the tradeoffs between the different algorithms are represented graphically in Fig. 8. Fig. 8(a) shows a phase diagram indicating which algorithm gives the minimum speedup S for particular values of T and N. The regions partitioned by the lines represent the parameters for which the labeled algorithm provides the smallest speedup. So, for small values of T, the MIN algorithm has the smallest speedup because it is the only algorithm for which $T > T_{\min}$. Soon after T is large enough for DOUBLE or LIST to be used, they become the algorithms of choice and likewise for the exact algorithm. For the example of $N = 128$ and $\delta = 200$, DOUBLE and LIST become preferred at approximately $T = 53\,200$ slot times and EXACT provides the lowest speedup at $T = 6\,400\,000$ slot times [marked as circles in Fig. 8(a)]. A similar graph is shown in Fig. 8(b) for the minimum delay T given S and N. As the speedup passes 2, DOUBLE and LIST become the favored algorithms, and at $4(4 + \log_2 N)$, MIN is preferred. In the example, DOUBLE and LIST provide the smallest delay at just beyond $S = 2$ and MIN at $S = 54.4$ [marked in Fig. 8(b)].

Finally, while the delays in this example may seem large for applications such as a packet switch, it is important to realize that they are within a small constant factor of the minimum possible delays for the switch size and reconfiguration overhead. To see this, consider the case when a cell arrives at each of the N input ports at the same time all destined to the same output. For an unconstrained switch, it is obvious that these cells can be transferred in N slot times, giving a delay of N slot times to the last cell. Similarly, a constrained switch in the same situation requires N slot times plus an additional $\delta(N - 1)$ slot times to reconfigure the switch between cells, giving a delay of approximately $(1 + \delta)N$ for the last cell. This simple example illustrates how a particular cell must incur an additional delay of $(1 + \delta)N - N = \delta N$ slot times when traveling through the constrained switch. Assuming $H = T$, the delays incurred by our architecture are $3\delta N$ for MIN and $6\delta N$ for DOUBLE and LIST, only slightly above the minimum.



(a)



(b)

Fig. 8. Algorithm phase diagrams over the design space. (a) Minimum speedup. (b) Minimum delay.

B. Average Case Performance of Algorithms

In the previous sections, only the worst case performance of the different scheduling algorithms has been considered. While the worst case is important for guaranteeing the correctness of the emulation architecture, average case performance of the algorithms may be of interest in systems that contain a mixture of best effort data with data that requires guarantees through the switch.

As described, the MIN and DOUBLE algorithms are not designed to optimize average case performance. So, for this section, modified versions of these algorithms, MIN* and DOUBLE* are used. In MIN*, the inner loop (Step 4) is only repeated for the actual number of colors required for G_A. Also, the weights in Steps 4b, 4c, and 6 are selected to be the maximum value of the corresponding scheduled elements from C instead of being the largest possible value. Finally, the outer loop (Steps 2–5) is repeated until exactly $N/4$ schedules have been produced. DOUBLE* contains a similar change: in Step 4 the weights are selected to be the maximum value required to cover the fine portion of the matrix.

For comparison, we also include GOPAL, the algorithm described in [8]. GOPAL first guarantees the minimum number of switchings $N_s = N$ and then tries to minimize schedule length by greedily considering the largest unscheduled elements of C. The algorithm is designed for average case performance and does not have a worst case guarantee.

Average case performance of the algorithms is determined by running multiple trials on randomly generated request matrices. Each request matrix $C(T)$ is created by summing T random permutation matrices, which are uniformly selected from the set of all permutations.

The average minimum speedup $S_{\min}$ (one over the fraction of empty slots) is shown versus the switch size N in Fig. 9(a) for $T = 1024$. Similar trends in both the average speedup and average delay are observed for other values of T. By definition, EXACT fills all the slots and, therefore, only requires $S_{\min} = 1$.

In contrast, MIN* leaves many empty slots and requires a speedup that steadily increases with N. However, the average case speedup for MIN* is significantly less than the worst case bound derived in Section IV. The sawtooth shape of MIN*'s speedup is caused by jumps in the number of iterations of the outer loop—each "tooth" in the graph corresponds roughly to the value of N for which one more iteration of Steps 2–5 can be performed.

DOUBLE*'s average speedup stays near 1.5 or approximately 75% of its worst case bound of 2. The oscillations as N increases beyond 100 are due to the ceiling function used to compute the weight in Step 3 of DOUBLE*: up to N schedules with total weight at most $N\lceil T/N \rceil$ are generated. The function $N\lceil T/N \rceil$ oscillates between 1 and 2 as N increases, which corresponds directly to the oscillations in the speedup required for DOUBLE*.

The speedup required for LIST has two distinct phases. Initially, the average speedup remains near optimal when the relative difference between entries in C tends to be small $(N \ll T)$. However, as N increases, the relative difference also increases, causing a jump in the speedup to approximately 1.5. For the case shown in Fig. 9(a), the transition between the two phases of operation occurs at approximately $N = 90$. As T increases, this transition point occurs at larger values of N.

Finally, GOPAL generally gives the best average case speedup excluding EXACT. It stays near optimal and grows only slightly as N increases.

Significantly less variation is found in the average minimum delays of the algorithms, which are shown in Fig. 9(b) normalized to δ. EXACT follows the $N^2 - 2N + 2$ curve necessary to prevent empty slots. By definition, the minimum switchings algorithms, MIN* and GOPAL, have normalized delays of N. While DOUBLE* can produce fewer than $2N$ schedules in the average case, the figure shows the average delay is only slightly below $2N$. However, LIST has close to optimal delay in the average case, generally using only a few more than N switchings.

In general, the two greedy scheduling algorithms, GOPAL and LIST, showed the best average case performance for uniform random request matrices.

VII. Related Work

The time-slot assignment problem has received significant attention in the context of scheduling satellite-switched time-division multiple access (SS/TDMA) systems. Notably, algorithms to find exact decomposition of a matrix C in a minimum number

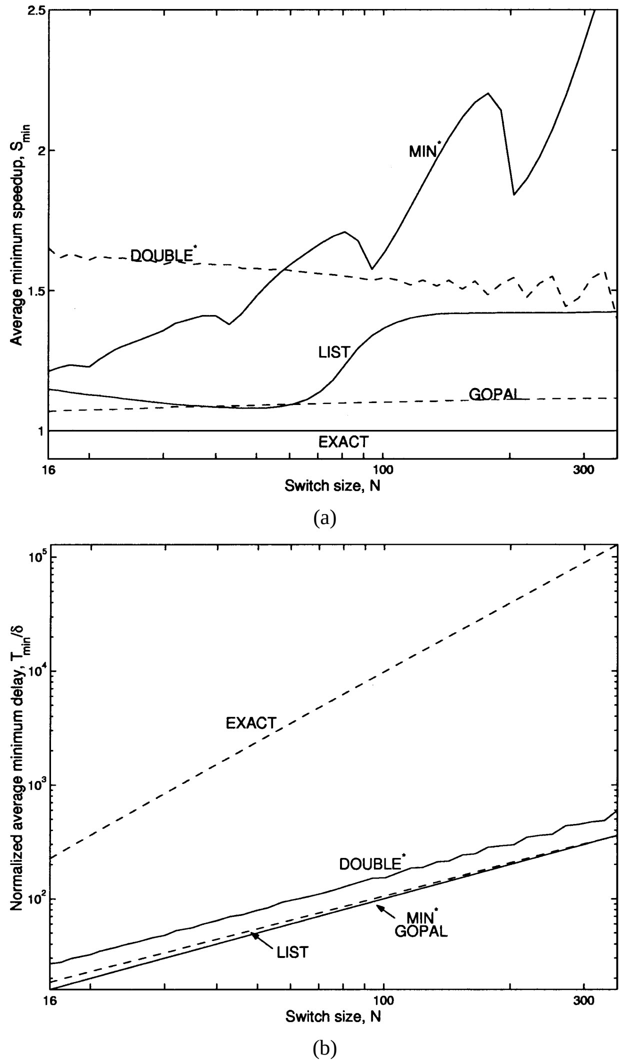


Fig. 9. Average case comparison of scheduling algorithms ($T = 1024$). (a) Average minimum speedup. (b) Average minimum delay.

of switch configurations are described in [7]. The idea of using only N switch configurations was introduced in [8], where the authors proved the problem of finding the minimum length schedule for a particular matrix C to be NP-complete. They also introduced a heuristic algorithm to create the schedules. The SS/TDMA scheduling problem is the same as the scheduling problem considered in this paper. However, making an analogy to packet routing, existing algorithms provide “best effort” schedules, where the goal is to minimize the average schedule length. We demonstrated new algorithms that solve the same scheduling problem, but have provable worst case guarantees necessary for emulation.

More recently, similar problems have been considered in wavelength-division multiplexing (WDM) systems. Both [17] and [18] provide heuristic algorithms for scheduling transmissions in star networks given a number of tunable receivers and transmitters with nonzero tuning latencies. Optimal all-to-all transmission schedules for the star networks are considered in [19]. The problems addressed by these researchers are more broad in that multiple transmitters and receivers per input are used, but again schedules are chosen to minimize the average length, not to provide a bounded worst case.

The impact of constrained switches on packet switch scheduling has also been addressed. The work of [20] develops an architecture and several algorithms to guarantee throughput and delay given a larger data envelope and, therefore, fewer logical switch configurations. This work complements our approach in that we develop techniques to implement a given number of logical switch configurations in fewer physical configurations, thus reducing the speedup requirements of the switch.

Also, as noted in both [21] and [22], the task of computing a schedule for an *unconstrained* switch is becoming a more difficult problem as switch sizes scale. Both of these papers provide solutions to this problem centered around decomposing a traffic matrix C into permutation matrices and show that the resulting switch is stable. The algorithms presented in this paper could readily be applied to this problem, extending the work of [21] and [22] to switches with nonzero switching overhead. [22] also notes that the exact scheduling algorithm’s requirement of $O(N^2)$ switch configurations limits scalability and proposes a multi-stage network to solve the problem. In this case in particular, the DOUBLE or LIST algorithms could also provide scalability for a speedup of 2.

As described in Section V, constrained switch scheduling can be cast as an open shop scheduling problem. It is a long-standing open question in operations research whether the factor of two is a necessary condition for list scheduling and also whether practical algorithms exist that outperform list scheduling. A polynomial time approximation scheme (PTAS)[6] for open shop scheduling is given in [23], but the authors admit the constant terms in the run-time of the algorithm make it impractical.

VIII. Conclusion

Optical switching technologies are becoming an attractive alternative to electronic switches as the demand for switch bandwidth and port count increase exponentially. However, many of these optical technologies have a large switching overhead, requiring from nanoseconds to milliseconds to reconfigure. Efficient scheduling of these constrained switches requires algorithms that consider this overhead.

We proposed an architecture and algorithms that allow a constrained switch to exactly emulate an unconstrained switch within a fixed delay. This decouples the task of accounting for configuration overhead from the traditional switch scheduling problem. Constrained switches can then be used directly in designs that can tolerate the fixed delay.

Providing emulation requires scheduling algorithms that have guaranteed bounds on the length of their schedules. We analyzed the speedup and delay required for emulation using three bounded algorithms across a range of port sizes N and batch sizes T. The EXACT algorithm provides the lowest speedup requirement, but is only attractive for very large batch sizes, which are needed to amortize the cost of its quadratic number of configurations, or very low port counts. We developed the MIN algorithm to use the minimum number of switchings $N_s = N$, but the speedup required was shown to be prohibitive, $\Theta(\log N)$. As a result, MIN is only attractive for small batch sizes, where it is the only algorithm that will work. Alternatively, our DOUBLE

[6]A PTAS approximates the optimal solution of a problem to within a factor of $(1 + \epsilon)$ for any $\epsilon > 0$.

algorithm balances a small number of switchings $N_s = 2N$ with a constant speedup of 2. DOUBLE offers the minimum required speedup across a wide range of N and T. The resulting family of algorithms provide a range of speedup versus delay tradeoffs, making emulation feasible over a large design space.

By allowing arbitrary reconfiguration of the switching elements, the constrained switching problem was then formulated as an open shop scheduling problem. A simple approximation algorithm LIST was shown to give the same speedup and delay requirements of DOUBLE. Interestingly, LIST gives the best known bounds for a practical open shop scheduling algorithm, so for our architecture there is no gain in adopting switching technologies that allow arbitrary reconfiguration.

The work presented here raises many interesting questions for future study. The algorithms we have presented represent several points in the space of N_s versus $S_{\min}$. It is interesting to ask what happens at other points. As we increase N_s from $2N$ to N^2 how rapidly does $S_{\min}$ fall from 2 to 1? Can a constant $S_{\min}$ be achieved for an N_s less than $2N$? Finally, while LIST is currently the best known practical approximation algorithm for open shop scheduling, it is still an open question whether a more efficient algorithm exists. For constrained switch scheduling, it may be possible to take advantage of the specifics of the problem to develop a tighter bound on the open shop schedule length.

APPENDIX

Two classical results from graph theory are used in the following sections.

Theorem 5: (Hall) For a bipartite graph $G = (X \cup Y, E)$, a perfect matching exists if and only if for all nonempty $S \subseteq X$, $|S| \leq |\mathcal{N}(S)|$ where $\mathcal{N}(S)$ is the set of vertices adjacent to S.

Theorem 6: (König) There exists an edge-coloring of any bipartite multigraph with a maximum degree of Δ which uses Δ colors.

A. Correctness of MIN

For simplicity, the MIN algorithm is presented for $N \geq 8$ and for this proof of correctness we also assume N is even.

Theorem 7: For a bipartite graph $G = (X \cup Y, E)$ with $|X| = |Y| = n$, there always exists a perfect matching in G if its minimum degree is greater than $n/2$.

Proof: Assume no perfect matching exists in the graph. Then by Hall's Theorem, there must exist a nonempty $S \subseteq X$ such that $|S| > |\mathcal{N}(S)|$. Since the minimum degree of G is greater than $n/2$, then $|S| > |\mathcal{N}(S)| > n/2$. Also, $|X - S| < n/2$.

By definition of $\mathcal{N}(S)$ there are no edges between $Y - \mathcal{N}(S)$ and S. Therefore, for any vertex $i \in (Y - \mathcal{N}(S))$, $\mathcal{N}(i) \subseteq (X - S)$. This implies $|\mathcal{N}(i)| \leq |X - S| < n/2$, which is a contradiction because the degree of i is greater than $n/2$. Therefore, G contains a perfect matching. □

Theorem 8: For a k-regular bipartite graph $G = (X \cup Y, E)$ with $|X| = |Y| = n$ and $k > 3n/4$, any partial matching M of G with $|M| \leq n/2$ is a subset of a perfect matching of G.

Proof: Construct a copy of G in G'. For each edge in M, remove the edge, its endpoints, and edges incident to those endpoints from G'. This leaves $2(n - |M|)$ vertices in G'. Also, each removal reduces the degree of the remaining vertices of G' by at most one. Therefore, the minimum degree of the remaining vertices of G' is at least $k - |M|$.

By Theorem 7, there is a perfect matching in G' if $k - |M| > (n - |M|)/2$, or rewriting, that $|M| < 2k - n$. From the Theorem statement, $|M| \leq n/2 = 2(3n/4) - n < 2k - n$, so there is a perfect matching M' in G'. If a vertex in G was not covered in the partial matching M, it was included in G' and must be covered in the perfect matching M'. Therefore, $M \cup M'$ is a perfect matching of G and M is a subset of this matching. □

Now the correctness of MIN can be examined step-by-step. Step 1 simply initializes the algorithm. Step 2 identifies all edges greater than T/d that have yet to be scheduled. The row (column) sums of A are less than d. Otherwise, the corresponding row (column) of C would be greater than $(T/d)d = T$, which is a contradiction because the row (column) sums of C are at most T. The graph G_A constructed in Step 3 has a maximum degree of at most $d - 1$ because the row (column) sums of A are less than d. Then, by König's Theorem, G_A can always be edge colored with $d - 1$ colors.

Now that all the edges have been identified in Step 2 and colored in Step 3, Step 4 loops over $d - 1$ colors, which is sufficient to visit each of the colors assigned to G_A. In Step 4a, half of the edges of a particular color are used as a partial matching in B. Since N is assumed to be even, $\lceil |M_c|/2 \rceil$ is at most $N/2$. By Theorem 8, Step 4a finds a perfect matching of G_B that includes E_a if G_B is k-regular with $k > 3N/4$. Regularity is enforced by the fact that only perfect matchings are removed from B throughout the algorithm. The condition on k is verified below. Also, it is possible that some of the edges in E_a were scheduled, and hence removed, since the coloring in Step 3. This is handled by simply removing these edges from E_a, which can only reduce $|E_a|$, ensuring the conditions of Theorem 8 still hold. Again, since N is even, there are at most $N/2$ edges remaining in M_c for Step 4b, so another perfect matching can be found. Therefore, Steps 4a and 4b together ensure that all the edges in G_A assigned to a particular color will be scheduled. Since this process is repeated over all the colors, all the edges in G_A will be scheduled during Step 4.

Once Step 5 is reached, all the entries greater that T/d have been scheduled during Step 4. So, during the next iteration, no entry will be greater than $2T/d$ (d has been updated in Step 5), which ensures the weight assigned to the schedules during Steps 4a, 4b, and 6 are sufficient to cover the corresponding elements of C. Also, since $2(d - 1)$ additional schedules are produced in each loop, the loop condition during Step 5 ensures the above constraint on k is met. Finally, the Step 6 extracts the remaining perfect matchings from B, which are guaranteed to exist because G_B is regular.

B. Correctness of DOUBLE

The row (column) sums of A, created in Step 1, are at most N

$$\sum_j a_{i,j} = \sum_j \left\lfloor \frac{c_{i,j}}{\frac{T}{N}} \right\rfloor \leq \frac{\sum_j c_{i,j}}{\frac{T}{N}} = N.$$

So, by König's Theorem, the edge-coloring produced during Step 2 uses at most N colors. Step 3 then produces at most N

schedules, using all the edges in A exactly once. Finally, Step 4 covers every entry uniformly using N more schedules, for a total of at most $2N$ schedules. Any entry (i,j) is covered $a_{i,j}$ times in Step 3 and once more in Step 4:

$$(a_{i,j}+1)\left\lceil \frac{T}{N} \right\rceil = \left(\left\lfloor \frac{c_{i,j}}{\frac{T}{N}} \right\rfloor + 1 \right) \left\lceil \frac{T}{N} \right\rceil \geq \frac{c_{i,j}}{\frac{T}{N}} \left(\frac{T}{N} \right) = c_{i,j}.$$

So, the schedules produced by DOUBLE cover every element of C.

Acknowledgment

The authors would like to thank B. Prabhakar for initial discussions on this problem, and P. Glynn and G. Weiss for their assistance with open shops. They also thank the reviewers and members of Concurrent VLSI Architecture group for their helpful comments and suggestions.

References

[1] A. Neukermans and R. Ramaswami, "MEMS technology for optical networking applications," *IEEE Commun. Mag.*, vol. 39, pp. 62–69, Jan. 2001.

[2] J. E. Fouquet *et al.*, "A compact, scalable cross-connect switch using total internal reflection due to thermally-generated bubbles," in *Proc. IEEE/LEOS Annu. Meeting*, Orlando, FL, 1988, pp. 169–170.

[3] L. Y. Lin, "Micromachined free-space matrix switches with submillisecond switching time for large-scale optical crossconnect," in *OFC Tech. Dig.*, 1998, pp. 147–148.

[4] O. B. Spahn *et al.*, "GaAs-based microelectromechanical waveguide switch," in *Proc. IEEE/LEOS Int. Conf. Optical MEMS*, 2000, pp. 41–42.

[5] A. J. Agranat, "Electroholographic wavelength selective crossconnect," in *1999 Dig. LEOS Summer Topical Meetings*, pp. 61–62.

[6] Y. Ito, Y. Urano, T. Muratani, and M. Yamaguchi, "Analysis of a switch matrix for an SS/TDMA system," *Proc. IEEE*, vol. 65, pp. 411–419, Mar. 1977.

[7] T. Inukai, "An efficient SS/TDMA time slot assignment algorithm," *IEEE Trans. Commun.*, vol. COM-27, pp. 1449–1455, Oct. 1979.

[8] S. Gopal and C. K. Wong, "Minimizing the number of switchings in a SS/TDMA system," *IEEE Trans. Commun.*, vol. COM-33, pp. 497–501, June 1985.

[9] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM J. Appl. Mathemat.*, vol. 17, no. 2, pp. 416–429, Mar. 1969.

[10] D. B. Shmoys, C. Stein, and J. Wein, "Improved approximation algorithms for shop scheduling problems," *SIAM J. Comput.*, vol. 23, pp. 617–632, 1994.

[11] D. S. Hochbaum, Ed., *Approximation Algorithms for NP-Hard Problems*. Boston, MA: PWS, 1997.

[12] D. M. Johnson, A. L. Dulmage, and N. S. Mendelsohn, "On an algorithm of G. Birkhoff concerning doubly stochastic matrices," *Canad. Math. Bull.*, vol. 3, pp. 237–242, 1960.

[13] L. Lovász and M. D. Plummer, *Matching Theory*. Amsterdam, The Netherlands: Elsevier, 1986.

[14] A. J. Hoffman and H. W. Wielandt, "The variation of the spectrum of a normal matrix," *Duke Math. J.*, vol. 20, pp. 37–39, 1953.

[15] R. Cole and J. Hopcroft, "On edge coloring bipartite graphs," *SIAM J. Comput.*, vol. 11, pp. 540–546, 1982.

[16] T. Gonzalez and S. Sahni, "Open shop scheduling to minimize finish time," *J. ACM*, vol. 23, pp. 665–679, 1976.

[17] M. Chen and T.-S. Yum, "A conflict-free protocol for optical WDMA networks," in *Proc. GLOBECOM*, 1991, pp. 1276–1281.

[18] A. Ganz and Y. Gao, "A time-wavelength assignment algorithm for a WDM star network," in *Proc. IEEE INFOCOM*, 1992, pp. 2144–2150.

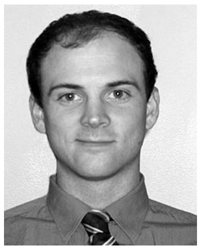
[19] G. R. Pieris and G. H. Sasaki, "Scheduling transmissions in WDM broadcast-and-select networks," *IEEE/ACM Trans. Networking*, vol. 2, pp. 105–110, Apr. 1994.

[20] K. Kar, "Reduced complexity input buffered switches," in *Proc. Hot Interconnects VIII*, 2000, pp. 13–20.

[21] E. Altman, Z. Liu, and R. Righter, "Scheduling of an input-queued switch to achieve maximal throughput," *Probabil. Eng. Inform. Sci.*, vol. 14, pp. 327–334, 2000.

[22] C. S. Chang, W. J. Chen, and H. Y. Huang, "Birkhoff-von Neumann input buffered crossbar switches," in *Proc. IEEE INFOCOM*, 2000, pp. 1614–1623.

[23] S. Sevastianov and G. Woeginger, "Makespan minimization in open shops: a polynomial time approximation scheme," *Mathemat. Program.*, vol. 82, pp. 191–198, 1998.



Brian Towles (S'99) received the B.Eng. degree in computer engineering from the Georgia Institute of Technology, Atlanta, in 1999 and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, in 2002. He is currently working toward the Ph.D. degree in electrical engineering at Stanford University.

His research interests include interconnection networks, network algorithms, and parallel computer architecture.

William J. Dally (S'78–M'86–SM'01–F'02) received the B.S. degree in electrical engineering from the Virginia Polytechnic Institute, Blacksburg, the M.S. degree in electrical engineering from Stanford University, Stanford, CA, and the Ph.D. degree in computer science from the California Institute of Technology (Caltech), Pasadena.

He and his group have developed system architecture, network architecture, signaling, routing, and synchronization technology that can be found in most large parallel computers today. While at Bell Telephone Laboratories, he contributed to the design of the BELLMAC32 microprocessor and designed the MARS hardware accelerator. At Caltech, he designed the MOSSIM Simulation Engine and the Torus Routing Chip which pioneered wormhole routing and virtual-channel flow control. While a Professor of electrical engineering and computer science at the Massachusetts Institute of Technology, Cambridge, his group built the J-Machine and the M-Machine, experimental parallel computer systems that pioneered the separation of mechanisms from programming models and demonstrated very low overhead synchronization and communication mechanisms. He is currently a Professor of electrical engineering and computer science at Stanford University, where his group has developed the Imagine processor, which introduced the concepts of stream processing and partitioned register organizations. He has worked with Cray Research and Intel to incorporate many of these innovations in commercial parallel computers, and with Avici Systems to incorporate this technology into Internet routers, and co-founded Velio Communications to commercialize high-speed signaling technology. He currently leads projects on high-speed signaling, computer architecture, and network architecture. He has published over 150 papers in these areas, and is an author of the textbook *Digital Systems Engineering* (Cambridge University Press, 1998).

Dr. Dally is a Fellow of the Association for Computing Machinery and has received numerous honors, including the ACM Maurice Wilkes Award.