Efficient Channel Scheduling Algorithms in Optical Burst Switched Networks

Jinhui Xu, Chunming Qiao, Jikai Li, and Guang Xu
Department of Computer Science and Engineering
State University of New York at Buffalo
201 Bell Hall
Buffalo, NY 14260, USA
{jinhui,qiao,jikaili,guangxu}@cse.buffalo.edu

Abstract— **Optical Burst Switching(OBS) is a promising paradigm for the next-generation Internet. In OBS, a key problem is to schedule bursts on wavelength channels whose bandwidth may become fragmented with the so-called void (or idle) intervals with both fast and bandwidth efficient algorithms so as to reduce burst loss. To date, only two scheduling algorithms, called Horizon and LAUC-VF, have been proposed, which trade off bandwidth efficiency for fast running time and vice versa, respectively.**

In this paper, we propose several novel algorithms for scheduling bursts in OBS networks with and without Fiber Delay Lines (FDLs). In networks without FDLs, our proposed *Min-SV* algorithm can schedule a burst successfully in $O(logm)$ time, where m is the total number of void intervals, as long as there is a suitable void interval. Simulation results suggest that our algorithm achieves a loss rate which is at least as low as the best previously known algorithm LAUC-VF, but can run much faster. In fact, its speed can be almost the same as Horizon (which has a much higher loss rate). In networks with FDLs, our proposed *Batching FDL* algorithm considers a batch of FDLs simultaneously to find a suitable FDL to delay a burst which would otherwise be discarded due to contention, instead of considering the FDLs one by one. The average search time of this algorithm is therefore significantly reduced from that of the existing sequential search algorithms.

I. Introduction

To meet the increasing bandwidth demands and reduce costs, several optical network paradigms have been under intensive research. Of all these paradigms, optical circuit switching (e.g., wavelength routing) is relatively easy to implement but lacks efficiency to cope with the fluctuating traffic and the changing link state; Optical Packet Switching(OPS) is a natural choice, but the required optical technologies such as optical buffer and optical logic are too immature for it to happen anytime soon. A new approach called Optical Burst Switching(OBS) that combines the best of optical circuit switching and optical packet switching was proposed in [1] [2], and has received an increasing amount of attention from both academia and industry worldwide [3], [5], [6], [7].

In an OBS network, an ingress OBS node assembles data (e.g., IP packets) into (data) bursts, and sends out a corresponding control packet for each burst. This control packet is delivered out-of-band and leads the burst by an offset time, o. The control packet carries, among other information, the offset time at the next hop, and the burst length l. At each intermediate node along the way from the ingress node to the egress node, the control packet reserves necessary resources (e.g., bandwidth on a desired output channel) for the following burst, which will be disassembled at the egress node.

A prevailing reservation protocol in OBS networks is called Just-Enough-Time (JET) whereby a control packet reserves an output wavelength channel for a period of time equal to the burst length l, starting at the expected burst arrival time r (which can be determined based on the offset time value and the amount of processing time the control packet has encountered at the node up to this point in time). If the reservation is successful, the control packet adjusts the offset time for the next hop, and is forwarded to the next hop. Otherwise, the burst is blocked and will be discarded if there is no Fiber Delay Lines (FDLs). If a FDL providing say d units of delay is available for use by the burst, and the channel will be available for at least l units of time starting at time $r + d$, the control packet will reserve both the FDL and the channel for the burst, which will not be dropped at this node.

Because bursts do not arrive one right after another, the bandwidth on each channel may be fragmented with the so-called “void” (or idle) intervals (see Figure 1). These void intervals may be utilized by a scheduling algorithm to make the reservation for some bursts whose corresponding control packets arrive after the void intervals have been created (which is possible when the JET protocol is used and the bursts have a variable, non-zero offset time). However, to keep the information on all existing void intervals and to search for a suitable one upon receiving a control packet (or equivalently a reservation request) could be a daunting task.

In OBS networks, a key problem is thus to design efficient algorithms for scheduling bursts (or more precisely their bandwidth reservation). An ideal scheduling algorithm should be able to process a control packet fast enough before the burst arrives, and yet be able to find a suitable void interval (or a suitable combination of a FDL and an void interval) for the burst as long as there exists one. Otherwise, a burst may be unnecessarily discarded either because a reservation cannot be completed before the burst arrives or simply because the scheduling algorithm is not smart enough to make the reservation.

Given the fact that OBS uses one-way reservation protocols such as JET, and that a burst cannot be buffered at any intermediate node due to the lack of optical RAM (a

0-7803-7753-2/03/$17.00 (C) 2003 IEEE

FDL, if available at all, can only provide a limited delay and contention resolution capability), burst loss performance is a major concern in OBS networks. Hence, an efficient scheduling algorithm that can reduce burst loss by scheduling bursts fast and in a bandwidth efficient way is of paramount concern in OBS network design.

So far, two well known scheduling algorithms have been proposed. Horizon [9] does not utilize any void intervals, and thus is fast but not bandwidth efficient (see Figure 2). On the other hand, LAUC-VF [3] can schedule a burst as long as it is possible but has a slow running time (see Figure 3).

In this paper, we propose an efficient way to organize the void intervals and as well as algorithms to schedule a burst as long as it is possible. Our algorithms can schedule bursts at least as efficiently as any existing scheduling algorithms (including LAUC-VF), and can handle the case with FDLs efficiently as well. In addition, our scheduling algorithms take as little as $O(\log m)$ time, where m is the total number of void intervals, which improves over the LAUC-VF algorithm by k times where k is the number of wavelengths on each link. (Note that since we can easily reduce the binary search problem to this channel scheduling problem, the lower bound of this problem is $O(\log m)$, meaning our algorithm is theoretically optimal.) In fact, our simulations show that on average, it can run as fast as Horizon (which has a much higher burst loss rate). Also, in case there are up to B different delays via FDLs available, our algorithm can batch process multiple possibilities and achieve an average running time which is much faster than that of existing approaches that sequentially checks one FDL at a time.

The rest of this paper is organized as follows. Section II provides a more detailed problem formulation and description of the existing Horizon and LAUC-VF approaches. Section III introduces the problem modeling and proposes several new scheduling algorithms, such as Min-SV and Batching FDL algorithm. We present simulation results and analysis in Section IV. Section V concludes our work.

II. Background

In this section, we first define the scheduling problem in more detail and then review the prior work on scheduling algorithms.

A. Problem Description

In an OBS network, it is possible that a control packet may arrive o units of time (which is called the offset time) before the corresponding burst b arrives. In such a case, the reservation for the burst will not start at the current time (t), but at $r = o + t$ (i.e., when the burst actually arrives). If the burst's length is l, the reservation will be made until $f = r + l$. Because bursts may not arrive one after another without any interval in between, each channel is likely to be fragmented with several reservation periods, separated by idle (also called void) intervals. More specifically, each of the k channels initially corresponds to a void interval from time 0 to positive infinite. Let each void interval I_j be modeled as an ordered pair (s_j, e_j), where s_j and e_j are the starting and ending time of the void interval I_j, respectively, with $e_j > s_j$. We say a void interval I_j is feasible to a data burst $b = (r, f)$, if and only if $s_j \le r$, and $e_j \ge f$. Once the reservation is made using a feasible interval I_j, up to two new void intervals may be created, which are (s_j, r) and (f, e_j), respectively.

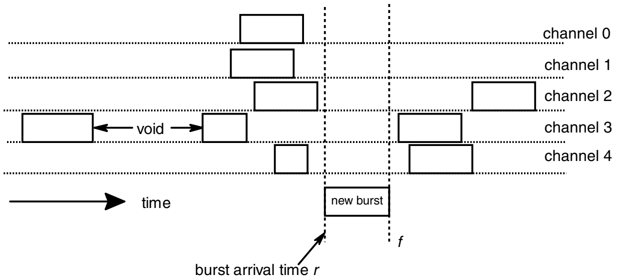

Fig. 1. Channels are fragmented into voids before scheduling a new burst.

An efficient scheduling algorithm should be able to fit a new reservation period into an existing void interval whenever possible to increase the bandwidth utilization and decrease the data loss rate.

The availability of FDLs at each node further complicates the design of scheduling algorithms. More specifically, assume that there are B different delays ($d_1 < d_2 < \cdots < d_B$) that a burst can obtain via FDLs at a node. Then, the possible offset time values are o, $o^1, o^2, \cdots, o^B$, where $o^j = o + d_j$ for $1 \le j \le B$. This in turns leads to $B + 1$ different starting times, r, $r^1, r^2, \cdots, r^B$, and finishing times f, $f^1, f^2, \cdots, f^B$ of the reservation period for the burst. An efficient scheduling algorithm thus needs to examine up to $B + 1$ possible ways to satisfy a reservation request for each burst.

In addition to be efficient in terms of bandwidth utilization and loss rate, a scheduling algorithm also needs to be fast as mentioned earlier. Assume that the minimum burst length is 1 unit time (e.g. a millisecond). For an OBS switching fabric having N input links, each multiplexed with k channels, the maximal number of control packets (or reservation requests) that may need to be processed is kN per unit time. For $N = 64$, $k = 100$ and a unit time of 1 millisecond, this translates to a required processing speed of 6.4 million requests per second.

B. Prior solutions and their limits

Several algorithms have previously been studied for solving the channel scheduling problem.

1) Turner designed the Horizon Scheduling algorithm [9]. In this algorithm, a scheduler only keeps track of the so-called horizon for each channel, which is the time after which no reservation has been made on that channel. The scheduler assigns each arriving data burst to the channel with the latest horizon as long as it is still earlier than the arrival time of the data burst (this is to minimize the void interval between the current horizon and the starting time of the new reservation period; see Figure 2 for an example). For a link with k channels, the best implementation of the horizon scheduling algorithm takes $O(logk)$ time to schedule a burst. Accordingly, the horizon algorithm is relatively simple and has a reasonably good performance in terms of its execution

time. However, the horizon scheduling algorithm results in a low bandwidth utilization and a high loss rate. This is due to the fact that the horizon algorithm simply discards all the void intervals.

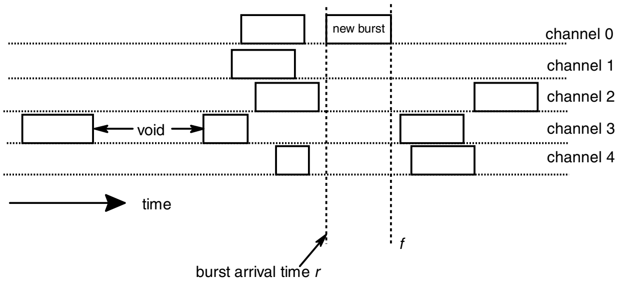

Fig. 2. Horizon schedules the new burst in Figure 1 to channel 0.

2) Xiong *et al.* [3] proposed a channel scheduling algorithm, called *LAUC-VF* (Latest Available Unused Channel with Void Filling). LAUC-VF keeps track of all void intervals (including the interval between the horizon and positive infinity), and assigns a burst arriving at time r a large enough void interval whose starting time s_i is the latest but still earlier than r. This yields a better bandwidth utilization and loss rate than the Horizon Algorithm. However, even the best known implementation of LAUC-VF has a much longer execution time than the Horizon Scheduling algorithm, especially when the number of voids m is significantly larger than k (which in general is the case). For example, a straightforward implementation of the LAUC-VF algorithm, described in [3] takes $O(m)$ time to schedule a burst. The time complexity becomes $O(Bm)$ when there are B different delays a burst can obtain via the use of FDLs. Searching for a suitable void interval in this way might take a longer time than that is allowed by the offset time of a burst, thus resulting in a failed reservation.

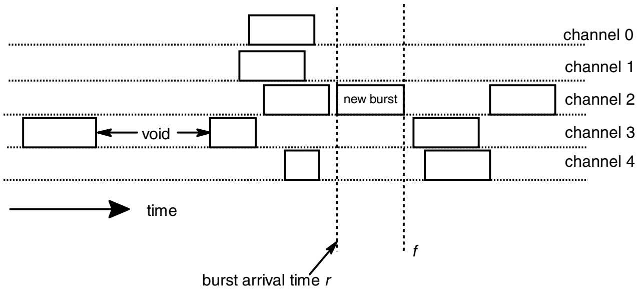

Fig. 3. LAUC-VF schedules the new burst in Figure 1 to channel 2.

III. Proposed Scheduling Algorithms

In this section, we present several efficient algorithms for selecting channels for incoming data bursts using different criteria. Our algorithms are based on interesting techniques from computational geometry.

A. Modeling the Problem Geometrically

We view each void interval I_j as a point with coordinates (s_j, e_j) on a two-dimensional plane whose x and y axes are the starting and the ending time respectively (see Figure 4). In addition, without considering the use of FDLs, the reservation period for each data burst b is mapped to a fixed point (r, f). When there is no ambiguity, we will also use I_j and b to denote the points (s_j, e_j) and (r, f), respectively.

Notice that since in each void interval the ending time is always larger than the starting time, all the void intervals are mapped to points above the 45° line $y = x$ (see Figure 4). Also, the set F_b of void intervals feasible to b lies in the unbounded region R which is to the left of the line $x = r$ and above the line $y = f$ (see Figure 5) (since each channel can have at most one void interval feasible to b, the total number of points inside R is at most k).

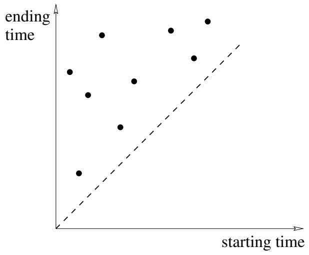

Fig. 4. Void intervals map to points.

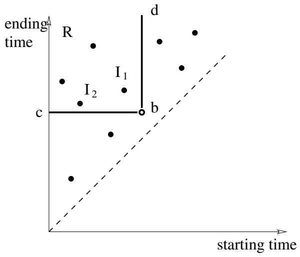

Fig. 5. Feasible region for data burst b.

If there is no point inside R, it means that no channel is available to the burst b with its current offset time o. In this case, if FDLs are available, one can consider using a FDL to effectively increase the offset time by a fixed value, and map the requested reservation period to a new point, say b^1, in the plane. In case there are B different delay times the burst can obtain, the desirable reservation periods correspond to a set of points, $b, b^1, b^2, ...,$ and b^B. All those points are on a straight line $y = x + l$ (where l is the duration of burst b). The feasible region for a data burst with the set of (B+1) offset times is bounded by a staircase curve from below (see Figure 6).

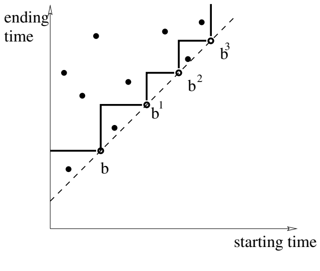

Fig. 6. Feasible region of a data burst b with multiple offset times.

B. Representative Criteria for selecting a channel

We have proposed scheduling algorithms that can apply several different criteria to select a channel for an arriving burst b. The first criterion is that for a given offset time, to find a void interval I_j which minimizes the difference between r and s_j among all feasible void intervals, i.e., $r - s_j = \min_{I_i \in F_b}(r - s_i)$. We call the feasible interval meeting this criterion as the *minimum starting void* or Min-SV fit, which aims to achieve the same objective as LAUC-VF (see Figure 3). Figure 5 shows an example, where I_1 is the Min-SV fit for b. Similarly, we can define the *minimum ending void* or Min-EV fit, which minimizes the difference between e_j and f (see Figure 5 for an example, where I_2 is the Min-EV fit for b; see also Figure 7 for another example), as well as their opposites, Max-SV fit and Max-EV fit, respectively.

Another criterion is called the *best fit* which finds a void interval I_j which minimizes the following (over all feasible void intervals I_i) $\min_{I_i \in F_b}\{(r - s_i) + (e_i - f)\}$. The weighted best fit finds a void interval to minimize the following weighted sum $\min_{I_i \in F_b}\{\alpha(r - s_i) + (1 - \alpha)(e_i - f)\}$ for $0 < \alpha < 1$.

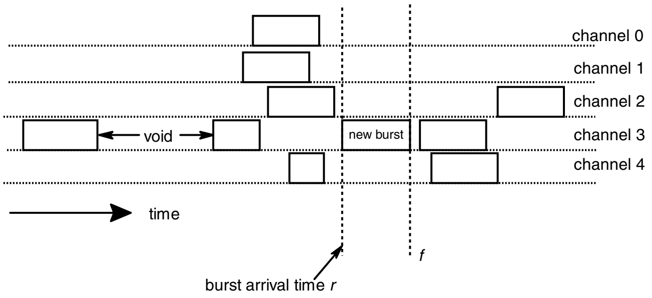

Fig. 7. Min-EV schedules the new burst in Figure 1 to channel 3.

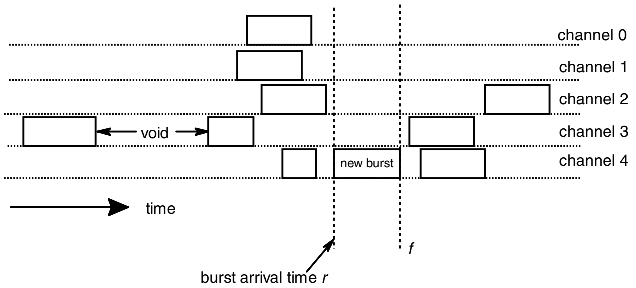

Fig. 8. Best-fit schedules the new burst in Figure 1 to channel 4.

C. Algorithm and Data Structure for the Min-SV fit

Below, we will describe the method to schedule a burst by finding a Min-SV fit which is the point in the feasible region R (in Figure 2) that is closest to the vertical line $x = r$. This method can easily be modified to find a Min-EV fit, Max-SV fit or Max-EV fit by rotating the coordinate system and/or reversing the order of searching for points in the coordinate system.

We first consider the case without any FDLs. We build a data structure, DS_{Min-SV}, to better organize all the void intervals. Our data structure is constructed by augmenting a balanced binary search tree (e.g., red-black tree [10]), which is can be viewed as a dynamic version of the priority search tree [11] supporting more restricted queries, and supports three major operations: burst query, interval insertion, and interval deletion.

The data burst query takes a burst b as input, and outputs the Min-SV interval, if such an interval exists. The interval insertion adds a new interval into the data structure, and the interval deletion removes a interval away from the data structure. To assign an available channel to a data burst b, one needs to first perform a burst query operation to find the Min-SV fit I_j, then a deletion operation to remove I_j from the data structure, and finally up to two insertion operations to insert up to two sub-intervals of I_j (called a starting void and an ending void, respectively) resulting from the channel assignment. Obviously, all three operations must be performed quickly.

To build the data structure DS_{Min-SV}, we first sort the set $\mathcal{I}$ of all void intervals by their starting times (this is done off-line first and incrementally thereafter), and then build a balanced binary search tree T_{start} based on the sorted starting times. This is accomplished by finding the interval I_m (called median interval) whose starting time s_m is the median of all the starting times in the considered set $\mathcal{I}$ of intervals. The median interval I_m is then used to partition $\mathcal{I}$ into two approximately equal-sized smaller sets, and each smaller set is partitioned recursively. All intervals in $\mathcal{I}$ will be the leaves of the search tree T_{start}. A non-leaf (or internal) node v in the tree corresponds to a vertical strip S^v in the coordinate system. More specifically, the root of the tree corresponds to the whole plane, and its two children correspond to the two half planes induced by the separating line $x = s_m$ crossing the median interval, and each half plane is recursively partitioned. Without loss of generality, the median interval is assumed to be in the left subtree.

Each internal node v is associated with the following information, s_m^v, p_{ymax}^v, and p_{ymin}^v, where s_m^v is the median starting time among all intervals in S^v, p_{ymax}^v is a pointer to the interval in S^v with the maximum ending time, and p_{ymin}^v is a pointer to the interval with the smallest ending time (the last value is used for clean-up operation to be described later).

To perform a burst query on T_{start} for a burst $b = (r, f)$, we can use the information stored in the internal nodes of T_{start} and perform the following procedure to search for the Min-SV fit.

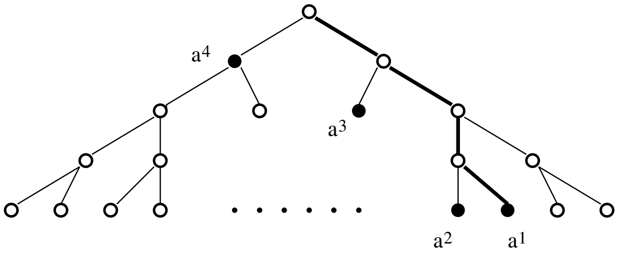

Fig. 9. Allocation nodes and search for Min-SV fit.

We first search the tree T_{start} to find all points to the left of the vertical line $x = r$ (some of which may be below the feasible region). To do this, we start at the root of the tree and compare r with the median starting time associated with the root. If r is larger, we mark the left child and proceed to the right. Otherwise, we simply proceed to the left (without marking the right child). The search then continues in a recursive fashion until a leaf node is reached. If the leaf node reached is a right child, as node a^1 in Figure 4, then its left sibling (also a leaf node) has to be marked. Otherwise, only the leaf itself (a left child) is marked.

We call the set A of marked nodes *allocation nodes* which correspond to vertical regions that contain intervals, or the intervals themselves whose starting times are no greater than r. The path P from the root of T_{start} to a^1 is called the *allocation path* – see Figure 9 for an example, where P is the dark curve, and A is the set of solid nodes. Note that, all the intervals whose starting time that is less than or equal to r are to the left of the leaf node reached in the end (i.e., a^1 in Figure 4). Thus to find the Min-SV fit for b, it is sufficient to check the marked leaf node and if needed, then search in the vertical strips corresponding to the marked non-leaf nodes.

To efficiently search for the Min-SV fit, we order all the nodes in A based on their height in T_{start}, and obtain a sequence $A = \{a^1, a^2, \cdots, a^h\}$, where $h = O(\log m)$ and m is the total number of intervals (or leaves) in the tree. In other words, we search for the Min-SV fit in a bottom-up order. It is easy to see that if a^1's ending time is no less than f, then a^1 is the Min-SV fit, so the algorithm can stop. Otherwise, we can check to see if a^2 is the Min-SV fit. Suppose we have searched all the nodes $a^1, a^2, \cdots, a^{j-1}$, and have not found the Min-SV fit yet. To determine whether the strip S^{a^j} corresponding to node a^j contains the Min-SV fit, we can first compare f (i.e., the finishing time of b) with the maximum ending time a^j_{ymax} (obtained by following the pointer $p^{a^j}_{ymax}$ stored at node a^j) in the strip S^{a^j}. If $f > a^j_{ymax}$, this means that none of the intervals in S^{a^j} is feasible to b, and hence the search in S^{a^j} can be terminated, and the next allocation node a^{j+1} should be checked. Otherwise, we know that the Min-SV fit will be definitely contained in S^{a^j} and therefore, no other allocation nodes need to be searched.

Thus our task now is to locate the Min-SV fit in the subtree of T_{start} rooted at a^j. For this purpose we first search the right child a^j_r of a^j, and check whether its associated strip $S^{a^j_r}$ contains the Min-SV fit (this can be done in constant time by comparing f with the maximum ending time in that region). If it does, our search can proceed recursively down the right subtree starting at a^j_r. Otherwise, the Min-SV fit must be in the left subtree, and the search will be done recursively there.

Note that a slightly faster implementation is to check the maximum ending time associated with an allocation node v before marking the allocation node for possible consideration later in the search for an Min-SV fit. The allocation node is marked if and only if the maximum ending time is larger than f (there can be at most k such marked nodes). In this way, the allocation node marked last (i.e., at the lowest level of the tree) is either the Min-SV fit itself (if it is a leaf) or the only one whose corresponding region needs to be searched for the Min-SV fit.

The running time of the searching procedure is $O(\log m)$. This is because the height of the tree is $O(\log m)$, and finding a^1 takes $O(\log m)$ time. There are at most $O(\log m)$ nodes in A, and checking the feasibility of each node a^j in A takes $O(1)$ time. Once a feasible node a^j of A is determined, we can recursively find the Min-SV fit in $O(\log m)$ time, since on each level of the subtree rooted at a^j, the search only spends $O(1)$ time. Thus the total time is $O(\log m)$. The total space occupied by this data structure is $O(m)$ as there are about $2m$ nodes in total (internal nodes or leaves) and each node needs $O(1)$ space.

For the interval insertion and deletion operations, it is clear that they can be done in $O(\log m)$ time, since inserting or deleting a node from a balanced binary search tree, like a red-black tree, takes $O(\log m)$ time, and the information stored in the internal nodes can be dynamically maintained in the same amount of time.

As discussed previously, to assign a void interval to an incoming data burst, one only needs to perform an interval query, and a constant number of interval insertions and deletions. Thus, using the above data structure, our scheduling algorithm can schedule an available channel for each incoming data burst in $O(\log m)$ time, and a sequence of n data bursts in $O(\log m(m+1)\cdots(m+n))$ time, which is at most $n\log(n+m)$ time, providing a significant improvement over the best previously known results.

Our data structure also supports a clean-up operation for removing those expired intervals (we call an interval expired if its ending time is smaller than the current time). Removing expired interval can reduce the size of the data structure and consequently improves the running time of any future query, deletion and insertion operations. The clean-up operation is implemented by using the pointer, p^v_{ymin}, stored in each internal node v. To clean up the data structure, we start at the root of T_{start}, and check whether there is any expired interval in the region associated it by comparing the current time with the minimum ending time in the region. If any expired interval is detected, we remove the interval pointed to by the pointer maintained at the root with a deletion operation. Clearly this can be done in $O(\log m)$ time for an expired interval.

To find a Max-SV fit, we can use the same data structure, but conceptually search the allocation nodes in a top-down fashion (i.e., from a^h to a^1). In practice, this can be done approximately 3 times faster than finding a Min-SV as only the region corresponding to the first marked allocation node needs to searched.

Similarly, to find a Min-EV fit, we construct a T_{end} tree, which is the same as T_{start} tree except the leaves are sorted according to their ending times in a descending order, and each internal node v contains the median *ending* time, and a pointer to the leave with the *smallest starting* time. The search algorithm first finds an allocation path using f as a key such that the leaves to the left all have their ending time larger than f. Once the allocation nodes are determined, search for an Min-EV fit starts at the bottom as in the search for an Min-SV fit described earlier. An Max-EV fit can be found by searching the allocation nodes in a top-down fashion.

Finally, one can find a random-fit by searching the region corresponding to a randomly selected marked node (and then if necessary, the regions corresponding to other marked nodes in a random order).

D. Scheduling Algorithm for the Case with FDLs

Scheduling a data burst with only one fixed offset time (i.e., without FDLs) may fail when there is no feasible void interval. One way to alleviate this problem is to use FDLs to effectively increase the offset time (and in turn, "shift" the reservation period).

Several approaches could be used to handle the case where there are B different delays that a burst may obtain via FDLs at a given node :

1) Run the Min-SV fit algorithm (or its variation) to schedule the data burst, starting with the minimum possible delay, until either the reservation succeeds or fails even with the maximum possible delay. This approach takes $O(B \log m)$ for scheduling a data burst in the worst case.
2) Select a set of $p \, (1 < p \leq B)$ different delays, and schedule the burst with any one of these delays in the associated feasible regions corresponding to the p possible reservation periods. This approach, as to be discussed next, essentially processes p possible delays in one batch, and has a shorter worst-case (and average case) running time than any previous approach with an appropriate value of p.

The main idea of the second approach, which we call the Batching FDL algorithm, is as follows.

To find a void interval for an incoming data burst b, the batching approach first uses either the Min-SV fit or the Min-EV algorithm to schedule a channel for b based on the original offset time (i.e., the offset time without the use of any FDLs). If such an interval exists, then it stops. Otherwise, a set $D = \{d_1, d_2, \cdots, d_p\}$ of p delays are selected, and a feasible void interval inside the union R_D of all the feasible regions for the corresponding reservation periods is sought. To speed up the search, the scheduling algorithm no longer looks for the Min-SV fit or the Min-EV fit. Instead, it reports the first fit I_1 found inside R_D by the search algorithm. This first fit may not require a minimum delay (i.e., the burst may be scheduled with a smaller delay than that is required by using this first fit). Note that as long as the delay is within an upper bound for a burst, introducing a non-minimum delay to the burst may not be bad as far as overall performance is concerned since other bursts may use the smaller delays that are still available to them.

Now the remaining difficulty is how to find a feasible void interval inside R_D. Our main idea is to try to find a "likely-feasible" void interval inside an enlarged region R'_D with simplified boundary, and trim the enlarged region down only when it is necessary. Such an idea comes from the observation that the time for searching an interval inside a region is proportional to the complexity (i.e., the number of vertices and edges) of the region. By enlarging and simplifying the feasible region, the algorithm can significantly reduce the average running time (to the point that it can be almost the same as that needed without FDLs), and improve the worst-case running time.

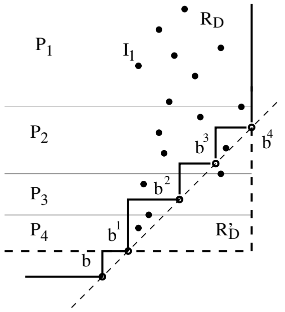

Fig. 10. Example of searching a batch of offset times.

More specifically, to locate a feasible interval in R_D, the scheduling algorithm first consider the enlarged region R'_D (see Figure 10 for an example), which is bounded by the vertical line $x = r^p$ from the right and by the horizontal line $y = f^1$ from the bottom, where r^p is the starting time of the data burst (after going through d_p units of delay), and f^1 is the ending time of the data burst (after going through d_1 units of delay). Clearly, R'_D is a superset of R_D, thus all the feasible intervals inside R_D are contained in R'_D.

To find a feasible interval in R_D, the algorithm first decomposes (searches) R'_D by finding its $O(\log m)$ *allocation nodes*(please refer to the definition in section III-C) in tree T_{end} (sorted according to the ending time), using $y = f^1$ as the search key (this results in an allocation path from the root all the way to the leave whose ending time is just slightly larger than f^1). Figure 10 shows an example where R'_D is decomposed into 4 subregions, P_1, P_2, P_3 and P_4, each associated with an allocation node along the allocate path from top to bottom).

Then, an interval with the smallest starting time within the topmost subregion (e.g., P_1 in Figure 10) is found (by following the pointer stored at the allocation node corresponding to the subregion), and if necessary, other subregions are then searched in a top-down approach. The intuition here is that the topmost subregion occupies a large portion of the space of R'_D with the smallest probability that it contains any steps (which is a part of the staircase corresponding to the shifted reservation periods). Also because the height of the corresponding allocation node in T_{end} is the highest, it is more likely that a feasible interval can be found in this region.

In general, if any interval I_j is detected in some subregion, say P_j, then the algorithm first checks whether I_j is inside R_D. This can be done by first determining the two finishing times f^q and f^{q+1}, where $1 \leq q < p$, such that they bound the ending time of I_j from below and above, respectively, if any. If the two values are found, and if the starting time of I_j is larger than r^q, I_j is outside R_D. Otherwise, it is inside R_D.

If I_j is inside R_D, the algorithm stops and reports the found interval, and uses the smallest delay possible, i.e., the smallest q such that r^q is larger than or equal to the starting time of I_j (note that $q \geq 1$ as it is assumed that the previous attempt to do without FDLs has failed already).

Otherwise, it means that P_j is not entirely contained in R_D

(e.g., P_2 in Figure 10), or in other words, there are some steps in P_j, and thus P_j needs to be further decomposed into subsubregions. This can be done by finding the allocation path in the subtree corresponding to P_j, using f^{q+1} as the search key. A better way is to use $f^{q'+1} \geq f^{q+1}$ as the search key, where $q' \geq q$ is the smallest value such that $r^{q'+1}$ is larger than the starting time of I_j. Intuitively, this limits searching the subsubregion (a horizontal strip) above line $y = f^{q'+1}$ (and below the region P_{j-1}). This is to avoid unnecessary search for intervals outside R_D as there is no feasible void interval to the left of I_j in the subregion P_j (as well as any subregions above) given I_j has the smallest starting time in subregion P_j. The algorithm is then recursively (or iteratively) applied to the subsubregion until either a feasible interval is found or it searches the entire subsubregion but still fails. In the latter case, the algorithm backtracks and searches subregion P_{j+1}, until either a feasible interval is found or it searches the entire region R_D but still fails.

Note that a subregion such as P_j needs to be decomposed only if it contains at least a step of the staircase (and an interval with the smallest starting time is found underneath the staircase), and the maximum number of times it needs to be recursively decomposed is equal to the total number of steps within the subregion. Also, it takes $O(\log(m/2^{l_j}))$ or $O(\log m - l_j)$ time each time to decompose subregion P_j, since the decomposition can be done in the subtree of T_{end} rooted at P_j, where l_j is the level of P_j in T_{end}. If it needs to be decomposed into subsubregions z times, the time to search the subregion P_j becomes $O(z \log m - z l_j)$.

Since at most p decompositions are needed to search the entire region R_D, the worst case time complexity of the Batching FDL algorithm for finding a feasible interval is less than $O(p \log m)$, since the level l_1 of P_1 is ≥ 1.

Once a feasible interval is found, deletion and insertion operations to schedule a burst can be done in $O(\log m)$ time. The clean-up operation can also be done in $O(\log m)$ time.

The average running time to find an feasible interval is much smaller. Part of the reason is that the algorithm stops whenever a feasible interval is encountered, and thus many regions containing those steps are not decomposed.

Once a feasible interval I_1 of b is located, assigning I_1 to b, which involves a constant number of interval insertions and deletions, can be done in $O(\log m)$ time. The additional storage requirement of this algorithm is $O(p + \log m)$.

E. Algorithm and data structure for finding the best fit

The objective of finding a best fit (or weighted best fit) is to minimize the total length (or weighted length) of the two void intervals (called starting void and ending void, and denoted by SV and EV, respectively) so as to further improve bandwidth utility and reservation success ratio.

Unlike the Min-SV fit in which the search criterion minimizes a distance in one dimension of the coordinate system (e.g., the starting time), the best fit criterion considers the L_1 distance (or Manhattan distance) which is the sum of a distance in the x dimension and a distance in the y dimension. See Figure 11, where $b = (r, f)$ is the data burst, and $I = (s, e)$ is a void interval in the feasible region R of b. The distance to be minimized is the sum of the lengths of two segments $sv = \overline{b \rightarrow I_y}$ and $ev = \overline{b \rightarrow I_x}$, where I_x is the horizontal projection of I on the vertical line $x = r$, and I_y is the vertical projection of I on the horizontal line $y = f$. The L_1 distance $sv + ev$ complicates the problem dramatically. Fortunately, the following observation can help us to simplify the problem.

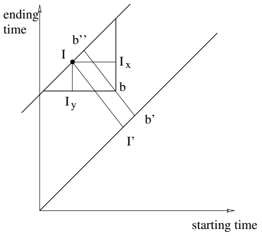

Fig. 11. Converting the bent distance to one dimension.

Observation 1: The L_1 distance is equal to the distance of $(|\overline{I \rightarrow I'}| - |\overline{b \rightarrow b'}|)$ times $\sqrt{2}$, i.e., $sv + ev = \sqrt{2} \times (|\overline{I \rightarrow I'}| - |\overline{b \rightarrow b'}|)$, where I' and b' are the orthogonal projections of I and b to the line $y = x$, respectively.

From this observation, and the fact that once a data burst is given, its projection distance $|\overline{b \rightarrow b'}|)$ is fixed, we know that to find the best fit of b, it is equivalent to find a void interval in the feasible region R such that its projection distance $|\overline{I \rightarrow I'}|$ is minimized.

Below, we show that we can solve the best fit query in $O(\log^2 m)$ time. Since the weighted best fit can be solved similarly by scaling up the y dimension by a factor of $(1 - \alpha)/\alpha$, we will only focus on the best fit.

Our data structure DS_B for computing the best fit for a data burst b makes use of the dynamic version of *range tree* [12] data structure. The basic idea is to construct a randomized balanced binary search tree [13] T_{end} based on the ending time of the void intervals, and for each internal node v of this tree, build another balanced binary search tree T^v_{start} based on the starting time for the intervals in the strip S^v associated with v only (i.e., not all the intervals). Notice that one can also use the starting time as the primary dimension.

Note that, different from the normal range-tree, our data structure does not apply the fractional cascading technique. This is because in OBS networks, the intervals are frequently inserted and deleted from the data structure, and the dynamic fractional cascading technique [14], although theoretically has a factor of $O(\frac{\log m}{\log \log m})$ improvement, is much more complicated and practically less efficient than our relatively simpler data structure.

To facilitate the search of the best fit in this data structure, in each node v of the tree T_{end}, we store the median interval I^v_{ym} (based on the ending time) and the minimum starting (and an optional minimum ending time) of the intervals in the strp S^v, and in each node u of T^v_{start}, we store the median interval I^u_{xm} (based on the starting time) and a pointer p^u_{pmin} to the interval whose *projection distance* is the minimum among all intervals in this subtree of T^v_{start} rooted at u.

To locate the best fit of b in this data structure, we first search on the tree T_{end}. Similar to the Min-SV fit case, we compute the set A_{end} of allocation nodes of b in T_{end}. We then search the best fit in the subtree rooted at each allocation node $a \in A_{end}$, and thus the actual best fit is the best among the $O(\log m)$ best fits we found from these allocation nodes.

Thus our focus now is on finding the best fit in an allocation node a. Instead of searching each subtree root at a in T_{end}, which is the case for computing the Min-SV fit, we search on the tree T^a_{start}. Note that the strip S^a associated with a in T_{end} is a horizontal strip whose y coordinate is $\geq$ the finishing time f of b. To find the best fit in S^a, we need to first find the feasible region in S^a (i.e., find the region whose x coordinate is $\leq$ the starting time r of b. This can be done by computing the set of allocation nodes A^a_{start} in the tree T^a_{start}. Each node $u \in A^a_{start}$ is thus associated with a rectangle S^u in the $x-y$ plane which is entirely contained in the feasible region R of b.

The remaining task is to find the best fit in the strip S^u. Recall that we have maintained a pointer p^u_{pmin} to the interval with the minimum projection distance in the subtree of T^a_{start} rooted at u. Hence, we can immediately obtain this interval by following the pointer.

Clearly the search for the best fit in each allocation node $a \in A_{end}$ can be done by searching $O(\log m)$ that many allocation nodes u's $\in A^a_{start}$ in the tree T^a_{start}. Each such allocation node u takes only $O(1)$ time for finding the local best fit. Hence, for each allocation node a, we need to spend $O(\log m)$ time, and since there are $O(\log m)$ allocation node $a's$ in A_{end}, the total time for searching the best fit is therefore $O(\log^2 m)$ time.

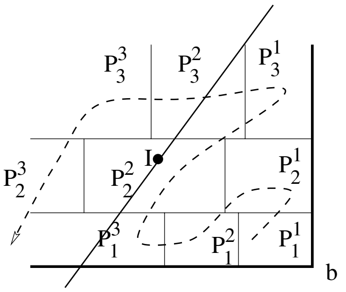

Fig. 12. Snake-search for the best fit.

An improved implementation of the above algorithm is to use the following snake-search procedure. Consider Figure 12, where the feasible region R is decomposed into three horizontal strips associated with three allocation nodes in T_{end}, and each strip is further partitioned into a set of rectangles which are entirely contained inside r, named as P^1_1, P^2_1, P^3_1,$\cdots$, P^1_2, P^2_2, P^3_2, $\cdots$, P^1_3, P^2_3, P^3_3, $\cdots$. Since our goal is to find an interval in R with the smallest projection distance, we can search these rectangles in an increased order of the projection distance (e.g., $P^1_1 \Rightarrow P^1_2 \Rightarrow P^2_1 \Rightarrow P^3_1 \Rightarrow P^2_2 \Rightarrow P^1_3 \cdots$). We call such a search fashion *snake-search*. When a local best fit $I = (s, e)$ is found by the snake-search in some rectangle (e.g., P^2_2), to ensure the optimality of I, we only need to further search those rectangles which intersect the halfspace HS_I bounded by the line $y = x + e - s$ from the above (see Figure 12). In this way, it is possible to avoid search many of the $O(\log^2 m)$ rectangles, and thus reduce the total running time. More specifically, after finding the allocation nodes A_{end} of b in T_{end}, we search these nodes in a bottom-up fashion. An allocation node $a \in A_{end}$ with higher height in T_{end} is searched (i.e., computing its allocation nodes A^a_{start} and then searching the associated regions) only when its corresponding region intersects the halfspace of some already found local optimum or no feasible iterval has been detected yet. The snake-search simultaneously seeks for the best fit in a set of secondary trees (i.e., T^a_{start} trees), all from bottom to top, and jumps from one to another in a snake fashion, until either the best fit is found or all of the $O(\log^2 m)$ rectangles have been searched.

To insert or delete an interval from this data structure, one need to first update the tree T_{end}, and then, for each node v whose subtree has been changed, update the tree T^u_{start}. Since the depth of T_{end} is $O(\log n)$, there are $O(\log n)$ secondary trees need to be updated, and each secondary tree takes amortized $O(\log m)$ time. The total updating time is amortized $O(\log^2 m)$. It is worth pointing out that all the information we stored in the nodes of tree T_{end} and the tree T^u_{start} can be dynamically maintained in the same amount of time for deletion and insertion.

To assign the best fit to an incoming data burst, one only needs to perform a query operation and a constant number of insertion and deletion operations. Thus the scheduling time for a burst is amortized $O(\log^2 m)$.

The clean-up operation in this data structure is similar to a deletion operation, and hence can be done in amortized $O(\log^2 m)$ time.

The storage (space complexity) required is $O(m \log m)$ because each leaf node in tree T_{end} can appear in at most $O(\log m)$ trees T_{start}.

Note that although experimental results on this algorithm are not available yet at the time when we submit this paper (experimental results will be included in the full version), we believe that the idea of minimizing the (weighted) sum of the lengths of the two void intervals will help us to simultaneously achieve better loss rate and a provably good scheduling time.

IV. Experimental Results

This section presents our experimental results on three of our algorithms, Min-SV, Min-EV, and Batching FDL, and their comparisons with existing algorithms, LAUC-VF and Horizon. Our experiments focus on examining the scheduling time and loss rate of these algorithms.

Our experiments are conducted on a Dell Precision 330 PC with a Pentium 4 CPU (1.3 GHz) and use Yacsim as the simulation tool. Our simulation observes the performance of one particular node in a network with two traffic sources. We assume that both burst length and control packet interarrival time follow a Pareto distribution, and the offset time of incoming bursts follows a uniform distribution in a certain range. The average burst duration is $1ms$.

In our simulations, we also consider several other parameters, such as the number of channels in a link (denoted by $ChanNum$), the offered link load (denoted by $LinkLoad$), the range of the offset time (denoted by $Range$), the number of FDLs (denoted by $FDLNum$), and the number of FDLs in a batch (called batch size, and denoted by p value). In our simulation, the scheduling time is obtained by measuring the time needed to schedule a large set of bursts and then taking the average.

A. Min-SV vs. LAUC-VF and Horizon

To compare our Min-SV algorithm with existing LAUC-VF and Horizon, we conduct two simulations, one investigates the relationship of scheduling time and the offered link load, and the other studies how the range of offset time affects the loss rate of the three algorithms,

The settings for the first simulation are: $ChanNum = 10$, and $Range = [0.3, 3]ms$. Simulation results in Figure 13 suggest that the change of link load nearly has no effect on the scheduling times of Min-SV and Horizon, which are always very small (close to 0). As to LAUC-VF, the simulation seems to suggest that the scheduling time is closely related to the link load. When the link load increases, the scheduling time grows almost linearly. Our simulation also demonstrates that the scheduling time of the three algorithms very much observes their time bounds predicted by the theoretical analysis, that is under any link load, Min-SV and Horizon take a much shorter time than LAUC-VF to schedule a burst.

For the case that the channel number is only 10, Min-SV has a slightly worse performance on the scheduling time than Horizon. We attribute this difference to the fact that Min-SV stores the whole set of the void intervals, whose cardinality could be much larger than the number of channels, while Horizon only needs to maintain in total 10 intervals. Furthermore, on each searching step, Min-SV needs to perform more operations (for maintaining the more complicated data structure) than Horizon.

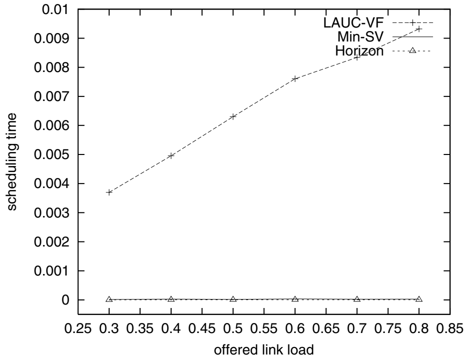

Fig. 13. Scheduling time vs. Link load.

It is also worth pointing out that the scheduling time of Min-SV remains a very small number even if the number of channels increases to hundreds. The scheduling time of LAUC-VF, however, can easily become unreasonably large when the channel number increases, thus preventing the simulation tool from yielding data for comparison.

The settings for the second simulation are: $ChanNum = 60$, $LinkLoad = 80\%$, and $Range = [\frac{a}{10}, a]ms$ at each comparison point. The simulation results, shown in Figure 14, indicate that the loss rate of Horizon algorithm is a fast growing function of the range of offset time, while that of the Min-SV and LAUC-VF almost remains the same when increasing the range. Our explanation on this phenomenon is the following. Horizon algorithm keeps only the last void interval for each channel and discards all other void intervals. With an enlarged range of offset time, the chance of generating void intervals in the time window between the current time and the starting time of the last interval on each channel increases, and therefore more bandwidth will be wasted, which consequently increases the loss rate. As for our Min-SV and LAUC-VF (actually the two algorithm have exactly the same loss rate), since the algorithms always keep all void intervals, and search the whole set of void intervals for each incoming burst, the loss rates are thus unlikely be affected by the increased range of offset time.

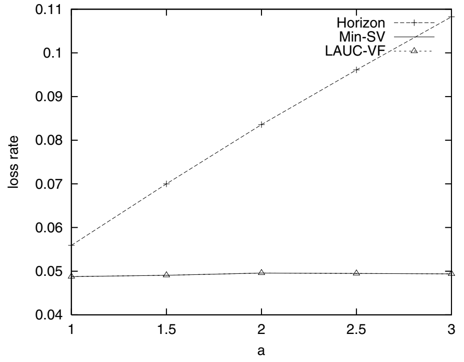

Fig. 14. Loss rate vs. Offset time range.

B. Min-SV vs. Min-EV

For Min-SV and Min-EV, we also conduct two simulations for studying their performance on loss rate and scheduling time under different link load.

Figure 15 shows the loss rate comparison between Min-EV and Min-SV under the following settings: $Range = [0.3, 3]ms$, and $ChanNum = 60$. The simulation results indicate that the loss rate of Min-EV is about 20% larger than that of Min-SV. This seems to suggest that one should try to minimize the starting void, instead of the ending void, as far as the loss rate is concerned with, because the left residual interval is more likely to be expired than the right one.

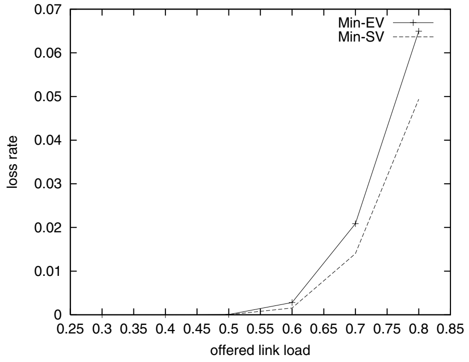

Fig. 15. Loss rate vs. Link load.

Figure 16 plots the scheduling time comparison of the two algorithms with $ChanNum = 60$ and $Range = [0.3, 3]ms$.

A surprising phenomenon discovered in this simulation is that Min-EV runs several times faster than Min-SV. Conceptually, Min-SV and Min-EV are symmetric to each other, seemingly suggesting that their scheduling time should be roughly the same. We attribute the significant difference to the fact that bursts are scheduled in a roughly increasing order of their ending times. When scheduling a burst $b = (r, f)$, the number of void intervals whose ending time is larger than f will be very small (i.e., no more than the number of channels plus the number of bursts who are scheduled before b, but have a larger ending time than f), and the feasible region R will therefore live in a very small subtree of T_{end} near the bottom of the T_{end}. Thus the total number of allocation nodes, as well as the search space can be dramatically reduced. Consequently, the scheduling time will be decreased.

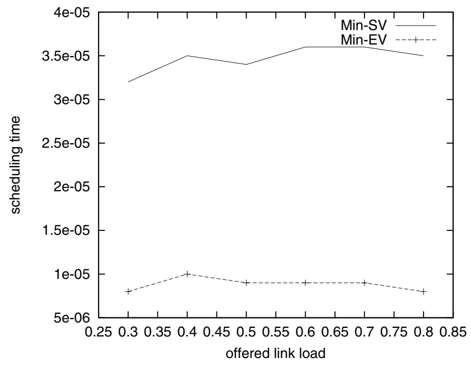

Fig. 16. Scheduling time vs. Link load.

The comparison between Min-SV and Min-EV demonstrates that minimizing different void could result in different performance on the scheduling time and loss rate. This suggests that a desired trade-off between scheduling time and loss rate might be achieved by choosing an appropriate α in the weighted best fit.

C. Batching FDL algorithm

To study the experimental performance of our batching FDL algorithm, we compare it with a sequential searching algorithm and study how the scheduling time and loss rate of the two algorithms change under different settings, such as different number of FDLs, and different batch size (i.e., the value of p). The sequential searching algorithm considers the available FDLs in an iterative manner, with each iteration using only one FDL to search for the feasible void interval, while the batching FDL algorithm considers a batch of FDLs as a whole and performs only one search on the set of void intervals.

In the comparison, both algorithms use Min-EV based algorithms to search for feasible void interval, and Horizon based algorithms for available FDLs. Since the time used for searching for the availabel FDLs could be significant comparing to the time used for searching for feasible void intervals, we use the following procedure to build an efficient data structure to facilitate the search for available FDLs.

- For each FDL, using Horizon algorithm to maintain the starting time (called horizon) of the last void interval.
- Partitioning the FDLs into groups such that FDLs in each group have the same delay.
- Using red-black tree to organize all the horizons in each group. For each node in the red-black tree, maintain a pointer to the minimum horizon in the subtree rooted at that node.
- Organizing all roots in another red-black tree.

The above procedure will result in a two-level red-black-tree structure which ensures that p FDLs can be found in $O(\log N + p)$ time, where N is the total number of FDLs with different delays.

Our first simulation is to study how the FDL number affects the scheduling time. The settings for this simulation are: $ChanNum = 60$, $Range = [0.3, 3]ms$, $p = FDLNum$, and $LinkLoad = 90\%$. Simulation results in Figure 17 show that the scheduling time of the sequential searching algorithm is significantly larger than that of the batching FDL algorithm, and their difference grows fast when the number of FDLs increases.

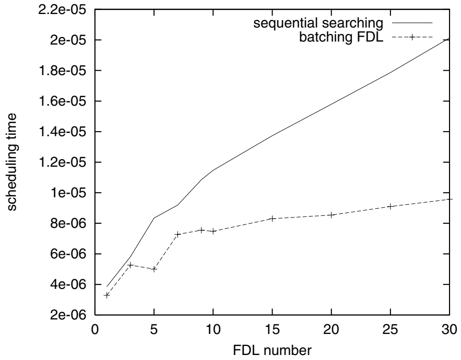

Fig. 17. Scheduling time vs. FDL number.

The second simulation compares the loss rate of the two algorithms using the same settings as in the first one. Figure 18 indicates that the loss rate of the batching FDL algorithm is higher than that of the sequential searching algorithm (but their difference increases only slowly with the number of FDLs). This is primarily because the batching FDL algorithm always picks the leftmost interval inside a region without optimizing the length of the starting or ending void, and thus it is likely to generate longer residual voids which are eventually wasted.

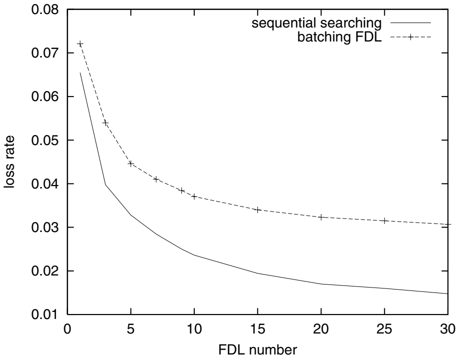

Fig. 18. Loss rate vs. FDL number.

Finally, simulations shown in Figure 19 and 20 exhibit how the batch size (i.e., the value of p) affects the scheduling time and loss rate. Both simulations have the following settings: $FDLNum = 30$, each FDL supports 30 channels, $ChanNum = 60$, $LinkLoad = 90\%$, Maximum delay time $= 3ms$. From the two simulations, we can expect that in general a better scheduling time and a lower lost rate of the batching FDL can be achieved by increasing the value of p. The reduced loss rate with larger p (and consequently larger delay) seems to suggest that it is better to distribute the bursts further apart in order to reduce the possibility of future contention on downstream links.

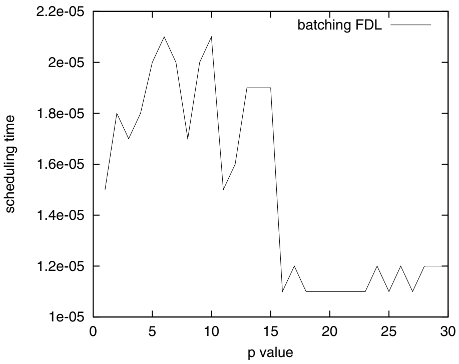

Fig. 19. Scheduling time vs. Batch size.

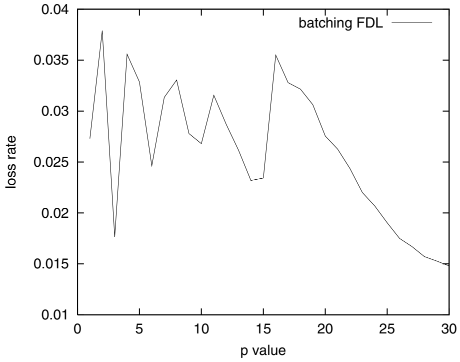

Fig. 20. Loss rate vs. Batch size.

V. Conclusion

In this paper, we have presented several novel channel scheduling algorithms (including Min-SV, Min-EV, Max-SV, Max-EV, Batching FDL, and Best Fit) in OBS networks using different channel selection criteria. Unlike existing channel scheduling algorithms, such as LAUC-VF and Horizon, which primarily aim at optimizing either the running time or loss rate, but not both, our algorithms take both performance metrics into consideration, and can perform well in networks with or without FDL. Most of our algorithms have a very shorter scheduling time for each incoming burst (e.g., worst case $O(\log m)$ time), and maintain a low loss rate.

We have implemented three of our algorithms, Min-SV, Min-EV, and Batching FDL, and conducted a comprehensive experimental study on them under different network settings (e.g., different channel number, different offered link load, and different offset time range). Particularly, we compared the running time and loss rate of our Min-SV algorithm with those of LAUC-VF and Horizon based on a large set (up to several millions) of randomly generated data bursts. Experiments have shown that our Min-SV runs much faster than LAUC-VF, and matches the running time of Horizon. As to the loss rate, our Min-SV algorithm has a much lower loss rate than Horizon, and the same loss rate as the LAUC-VF algorithm. We have also compared our Batching FDL algorithm with a sequential searching algorithm and found that our Batching FDL algorithm significantly improves the running time of the sequential searching algorithm, and suffers only a minor increase ($< 1.6\%$) in the loss rate in a heavily loaded network. Finally, we note that although the algorithms have been presented in the context of OBS networks, they are equally effective in many practical systems with limited resources, where advance reservations, each for a fixed period of time, need to be made.

References

[1] M. Yoo and C. Qiao, "A high speed protocol for bursty traffic in optical networks," *SPIE's All-Optical Communication Systems:Architecture, Control and Protocol Issues*, vol. 3230, pp. 79–90, Nov, 1997.

[2] C. Qiao and M. Yoo, "Optical burst switching(obs)-a new paradigm for an optical internet," *Journal High Speed Networks*, vol. 8, pp. 69–84, 1999.

[3] Y. Xiong, M. Vandenhoute, and H.C. Cankaya, "Control architecture in optical burst-switched wdm networks," *IEEE Journal on Selected Areas in Communications*, vol. 18, pp. 1838–1851, 2000.

[4] J.Turner, "Terabit burst switching," *Journal High Speed Networks*, vol. 8, pp. 3–16, 1999.

[5] L. Xu, H. Perros, and G. Rouskas, "Techniques for optical packet switching and optical burst switching," *IEEE Communications Magazine*, vol. 39,no.1, pp. 136–142, Jan. 2001.

[6] Andrea Detti and Marco Listanti, "Impact of segments aggregation on tcp reno flows in optical burst switching networks," in *IEEE Infocom 2002*, pp. 1803–1812.

[7] Ching-Fang Hsu, Te-Lung Liu, and Nen-Fu Huang, "Performance analysis of deflection routing in optical burst-switching networks," in *IEEE Infocom 2002*, pp. 66–73.

[8] C. Qiao, "Labeled optical burst switching for ip-over-wdm integration," *IEEE Communications Magazine*, vol. 38, no.9, pp. 104–114, 2000.

[9] J. Turner, "Terabit burst switching progress report (9/98-12/98)," in *Washington University at St.Louis Technical Report*, 1998.

[10] T.H. Cormen, C.E. Leiserson, and R.L.Rivest, *Introduction to Algorithms*, McGraw-Hill, MIT Press, 1990.

[11] E. McCreight, "Priority search trees," *SIAM J. Computing*, vol. 14, No.2, pp. 257–276, 1985.

[12] F. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[13] Conrado Martinez and Salvador Roura, "Randomized binary search trees," in *Research report of Universitat Politcnica de Catalunya, LSI-97-8-R*, 1997.

[14] K. Mehlhorn and S. N αher, "Dynamic fractional cascading," *Algorithmica*, vol. 5, pp. 215–241, 1990.