TR-IIS-05-017

Aggressive Traffic Smoothing for Online Delivery

Jeng-Wel Lin, Ray-I Chang, Jan-Ming Ho, Feipei Lai

September 2005 || Technical Report No. TR-IIS-05-017
http://www.iis.sinica.edu.tw/LIB/TechReport/tr2005/tr05.html

Aggressive Traffic Smoothing for Online Delivery

JENG-WEI LIN[1], RAY-I CHANG[2], JAN-MING HO[1], FEIPEI LAI[3]

**[1]Institute of Information Science,
Academia Sinica, No. 128, Sec. 2, Academia Rd., Taipei, Taiwan
{jwlin,hoho}@iis.sinica.edu.tw**

**[2]Dept. of Engineering Science,
National Taiwan University, No.1, Sec. 4, Roosevelt Rd., Taipei, Taiwan
rayichang@ntu.edu.tw**

**[3]Dept. of Computer Science and Information Engineering,
National Taiwan University, No.1, Sec. 4, Roosevelt Rd., Taipei, Taiwan
flai@cc.ee.ntu.edu.tw**

Abstract

Traffic smoothing is an efficient means to reduce the bandwidth requirement for transmitting a VBR video. Several traffic smoothing algorithms have been presented to offline compute the transmission schedule for a prerecorded video [1]-[9]. For live video applications, Sen et al. [11] present an online algorithm referred to as *SLWIN*(k) to compute the transmission schedule on the fly. *SLWIN*(k) looks ahead W frames to compute the transmission schedule for the next k frametimes, where $k \leq W$. Note that W is upper bounded by the initial delay of the transmission schedule. The time complexity of *SLWIN*(k) is $O(W*N/k)$ for an N frame live video. In this paper, we present an $O(N)$ online traffic smoothing algorithm denoted as *ATS* (Aggressive Traffic Smoothing). *ATS* aggressively works ahead to transmit more data as early as possible for reducing the peak rate of the bandwidth requirement. We compare the performance of our algorithm with *SLWIN*(k) based on several benchmark video clips. Experiments show that *ATS* further reduces the bandwidth requirement, especially for interactive applications in which the initial delays are small.

Keywords: Multimedia Streaming, Online Delivery, Traffic Smoothing, Live Video.

I. Introduction

A growing number of applications such as digital libraries, newscasts and distance learning require real-time multimedia to be accessed on networks. For supporting continuous playback, the client player must render a new frame after one *frametime* (the period of time between two successive frames) has passed. In order to prevent the client player from starvation, the video server has to transmit the video data into the client buffer before it is going to be rendered. However, video streams are compressed and often exhibit significant burstiness of *frame sizes* (number of bits for each frame) on many time scales due to the encoding frame structure and their natural variations within and between scenes [13][14]. This burstiness complicates the design of a multimedia system for high resource utilization, such as network bandwidth and the client buffer.

For a prerecorded video, the video server has complete knowledge of all the frame sizes. The video server can use a traffic smoothing algorithm that takes advantage of the knowledge of upcoming large frames and starts more data transmission in advance of the burst. By working ahead, traffic smoothing is proved efficient at reducing the bandwidth requirement for VBR video transmission. In past years, several traffic smoothing algorithms have been presented to offline compute the transmission schedule [1]-[9].

In live video applications, however, the video server only has limited knowledge of frame sizes at any one time. Old media data is transmitted, while new media data is generated simultaneously. In applications like newscasts and distance learning, the clients may be willing to tolerate a longer playback delay in exchange for a smaller bandwidth requirement. For such delay tolerable applications, Sen et al. [11] introduce a sliding-window traffic smoothing algorithm referred to as *SLWIN*(k) that computes the transmission schedule on the fly. The constant k is referred to as the slide length. Given an initial delay, *SLWIN*(k) looks ahead a window of W frames and uses an offline traffic smoothing algorithm to compute the transmission schedule for the next k frametimes, where $k \leq W$. Note that W is upper bounded by the initial delay. After k frametimes have passed, k new frames have to be generated, so *SLWIN*(k) computes the next transmission schedule. For an N frame live video, the time complexity of *SLWIN*(k) is $O(W*N/k)$. Since the time-consuming *SLWIN*(1) usually computes the transmission schedule of a small peak bandwidth requirement, there is a tradeoff between computing costs and performance.

In this paper, we present an $O(N)$ traffic smoothing algorithm called *ATS* (Aggressive Traffic Smoothing) that online computes the transmission schedule for live video applications. *ATS* uses a funnel with one upper chain and one lower chain to maintain the candidates for the next transmission schedule. It considers each new frame iteratively and modifies the funnel. If the two chains of the funnel will cross each other, *ATS* deterministically generates the transmission schedule. When no more frame size information is available, *ATS* heuristically generates the transmission schedule that works ahead as aggressively as possible without raising the current peak transmission rate. While the video server executes the transmission schedule, new frames are generated and after the schedule has finished, *ATS* computes the next schedule. Experiment results show that *ATS* further reduces the peak bandwidth requirement and utilizes the client buffer more efficiently when the initial delay is small.

The remainder of this paper is organized as follows. A formal definition of the online traffic smoothing problem for live video is illustrated in Section II. Section III presents our *ATS* algorithm and its time complexity proof. Section IV shows the experiment results. Finally, in section V, we give conclusions.

II. Online Traffic Smoothing for Live Video

For an N frame video $V=\{f_0, f_1, f_2, \ldots, f_{N-1}; T_f\}$, which uses f_i bits to encode the i-th frame and T_f is the frametime, a continuous playback schedule can be represented as a step function $F(t)=F_i$

for $i*T_f \leq t < (i+1)*T_f$, where $F_i=0$ for $i<0$ and $F_i=F_{i-1}+f_i$ for $0 \leq i \leq N-1$. At time $i*T_f$, the client player will have played F_{i-1} bits and will continue playing f_i bits in the next frametime. Given a D frametime playback delay, a video server can transmit media data at rate r_i from time $i*T_f$ to $(i+1)*T_f$ according to a transmission schedule $S=\{r_{-D}, r_{-D+1}, r_{-D+2}, \ldots, r_{N-2}\}$. The cumulative transmission function is defined as $G(t)=0$ for $t \leq -D*T_f$ and $G(t)=G(i*T_f)+r_i*(t-i*T_f)$ for $i*T_f < t \leq (i+1)*T_f$. To guarantee continuous playback at the client player, S should satisfy $F(t) \leq G(t)$ for $t \leq (N-1)*T_f$. However, the client player usually does not have unlimited buffer space. We assume the client player provides a B-bit buffer. Therefore, to avoid buffer overrun, S should also satisfy $G(t) \leq H(t)$ for $t \leq (N-1)*T_f$, where $H(t)=F_{i-1}+B$ for $i*T_f < t \leq (i+1)*T_f$, as shown in Figure 1.

Without loss of generality, we consider a discrete time model at the granularity of a frametime. Assuming $T_f=1$, we simplify the definition of $F(t)$, $G(t)$ and $H(t)$ as follows.

Cumulative playback function:

$F(i)=0,$	$i<0$
$F(i)=F(i-1)+f_i,$	$0 \leq i \leq N-1$

Cumulative buffer function:

$H(i)=F(i-1)+B,$	$\mathrm{i} \leq N-1$

Cumulative transmission function:

$G(i)=0,$	$i \leq -D$
$G(i)=G(i-1)+r_{i-1},$	$-D < i \leq N-1$

As $F(N\text{-}1)$ is the total size of the video frames, a traffic smoothing algorithm should not plan to transmit more than $F(N\text{-}1)$ data. For a prerecorded video stream, a traffic smoothing algorithm knows each frame size and can offline compute the transmission schedule such that $F(i) \leq G(i) \leq Min(H(i), F(N\text{-}1))$ for $-D \leq i \leq N\text{-}1$.

However, for live video, a traffic smoothing algorithm has limited knowledge of frame sizes at any one time $t+D$ because frames after the time point $t+D$ have not been generated. The algorithm knows $f_{t+1}, f_{t+2}, \ldots, f_{t+D}$ and has no idea about $f_{t+D+1}, f_{t+D+2}, \ldots, f_{N\text{-}1}$. Any L $(1 \leq L \leq D)$ frametimes transmission schedule $S^t = \{r_t, r_{t+1}, \ldots, r_{t+L\text{-}1}\}$ is feasible if S^t satisfies $F(i) \leq G(i) \leq Min(H(i), F(t+D))$ for $t < i \leq t+L$, as shown in Figure 2. While the video server executes S^t, new frames are generated. After L frametimes have passed and S^t has therefore finished, L new frames have been generated. The traffic smoothing algorithm can incorporate new frame size information to compute the next transmission schedule S^{t+L}.

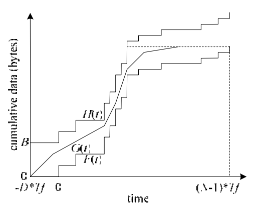

Figure 1: A feasible transmission schedule S should satisfy $F(t) \leq G(t) \leq H(t)$, where $G(t)$ is a function of S.

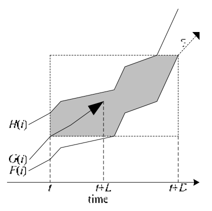

Figure 2: Looking ahead in live video applications.

A. Hopping-Window Approach

The hopping-window approach is quite simple. In the beginning, the video server waits for

W (*W*≤*D*) frames and uses an offline smoothing algorithm to compute the transmission schedule. After the transmission schedule has finished and *W* new frames have to be generated, the video server computes the next schedule.

The hopping-window approach splits the entire live video transmission into several *W* frame windows. However, it has drawbacks of smoothing traffic across window boundaries. As shown in Figure 3(a), since the smoothing algorithm does not know the upcoming of the large frame in the third window, it computes a low transmission rate in the second window. It is forced to sharply raise the transmission rate in the third window.

B. Sliding-Window Approach

To incorporate the new frame size information as early as possible, a sliding-window approach referred to as *SLWIN*(*k*) [11] is introduced. After *k* (k≤*W*) frametimes have passed, although the pervious transmission schedule has not yet finished, *SLWIN*(*k*) computes a new schedule. The video server immediately abandons the previous transmission schedule and executes the new schedule. The constant *k* is referred to as the slide length.

As shown in Figure 3(b), *SLWIN*(*k*) computes a better transmission schedule for the fourth window than the hopping-window approach. Sen et al. showed that using a smaller slide length, *SLWIN*(*k*) computes a better transmission schedule [11]. However, the total time complexity of *SLWIN*(*k*) is $O(W*N/k)$. There is a tradeoff between computing costs and performance.

C. Aggressive Workahead Scheme

SLWIN(*k*) uses an offline smoothing algorithm to compute the smallest bandwidth requirement for each window independently. However, *H*(*i*) is suppressed by *F*(*t*+*D*), as shown in Figure 2. This local optimum may lower the transmission rate unnecessarily. As shown in

Figure 3(b), the transmission rate decreases in the third window and then increases in the fourth window.

Instead of using the smallest transmission rate, the video server may aggressively works ahead. As there is less data buffered in the video server, the server can use a lower transmission rate to transmit the upcoming large frame. As shown in Figure 3(c), if the video server aggressively uses the transmission rate for the second window to transmit data in the third window, the bandwidth requirement for the fourth window can be reduced.

To reduce the *suppression effect* caused by *F*(*i*+*D*), we propose the video server to work ahead as aggressively as possible without raising the peak transmission rate. Since the suppression will be looser after a new frame is generated, aggressive workahead lasts for only one frametime. Experiment results show that aggressive workahead further reduces the peak bandwidth requirement, especially when the initial delay is small.

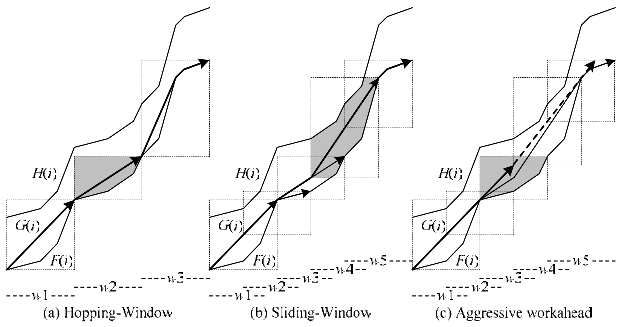

Figure 3: (a) The hopping-window algorithm has drawbacks when smoothing traffic across window boundaries. (b) *SLWIN*(*k*) incorporates new frame size information and therefore computes a better transmission schedule. (c) If the video server transmits video data in the third window move aggressively, the bandwidth requirement for the fourth window can be further reduced.

III. Our Algorithm and Time Complexity Analysis

A. *ATS* Algorithm

Like *MVBA* [1] (the offline smoothing algorithm used by *SLWIN*(k)), our *ATS* algorithm finds the shortest path that comprises of line segments (i.e., periods of a steady transmission rate) between $F(i)$ and $MIN(H(i), F(i+D))$. However, if the path is suppressed by $F(i+D)$, *ATS* heuristically uses the aggressive workahead scheme to generate the transmission schedule.

As shown in Figure 4(a), from the starting point $s=(t, G(t))$ of a window, *ATS* maintains the candidates for transmission schedules within a convex upper chain $U=\{u_0=s, u_1, u_2,\ldots, u_m\}$ and a concave lower chain $V=\{v_0=s, v_1, v_2,\ldots, v_n\}$. $u_1, u_2,\ldots, u_m$ are points on the function H and $v_1, v_2,\ldots, v_n$ are points on the function F. U is convex if and only if $0\leq m<2$ or $Rate(u_i, u_{i+1})<Rate(u_{i+1}, u_{i+2})$ for $0\leq i<m-2$, where $Rate(x, y)$ is the slope of the line from x to y. V is concave if and only if $0\leq n<2$ or $Rate(v_i, v_{i+1})>Rate(v_{i+1}, v_{i+2})$ for $0\leq i<n-2$. The two chains form a funnel. By triangle inequality, it is easy to prove that the shortest path is in the funnel.

For $t+1\leq a\leq t+D$, *ATS* iteratively considers unprocessed $F(a)$ and $H(a)$ and modifies the funnel. *ATS* first considers to append the point $x=(a, F(a))$ onto V. *ATS* may remove some points from V so that V is still concave. If the resultant V will not cross U, as shown in Figure 4(b), *ATS* continues processing $H(a)$. If the resultant V will cross U, as shown in Figure 5(a), *ATS* deterministically generates the transmission schedule according to the line segments on U that are under the dashed line (s, x) and maintains the funnel, as shown in Figure 5(b). By triangle inequality, we can prove that the path that comprises these line segments is a part of the shortest path.

If $H(a)\leq F(t+D)$, *ATS* then considers to append $x'=(a, H(a))$ onto U. Similarly, *ATS* may remove some points from *U* so that *U* is still convex. If the resultant *U* will not cross *V*, as shown in Figure 4(c), *ATS* continues processing the next frame. If the resultant *U* will cross *V,* as shown in Figure 5(c), *ATS* deterministically generates the transmission schedule according to the line segments on *V* that are above the dashed line (s, x') and maintains the funnel, as shown in Figure 5(d). Again, we can prove that the path that comprises these line segments is a part of the shortest path.

In each window, whenever *ATS* generates the transmission schedule, it goes to sleep. After the transmission schedule has finished, new frames have to be generated and *ATS* will wake up to compute the schedule for the next window. Note that the length of the transmission schedule is dynamic. However, saturation may occur if *ATS* does not decide the transmission schedule after the (t+D)-th frame is considered, as shown in Figure 5(e). Unlike *MVBA*, which always generates the transmission schedule according to *V, ATS* uses the aggressive workahead scheme to reduce the suppression effect. It generates the transmission schedule according to the dashed line (s, x''), where $x''=(t+1,\ G(t)+MAX(MIN(r_{peak},\ r_{max}),\ r_{min}))$, r_{min} and r_{max} are the minimal and maximal feasible transmission rates from the point s and r_{peak} is the current peak transmission rate. Note that x'' is definitely in the funnel and the length of the transmission schedule is one frametime. *ATS* then reconstructs the funnel again by removing some points and adding x'' at the head of *V* and *U*, as shown in Figure 5(f). *ATS* then goes to sleep. After one frametime has passed, *ATS* will wake up to compute the next transmission schedule.

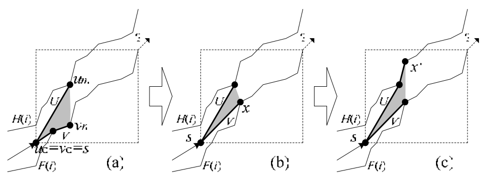

Figure 4: *ATS* iteratively considers *F*(*i*) and *H*(*i*) to maintain the candidates for transmission schedules in the funnel, i.e. the shadowed area.

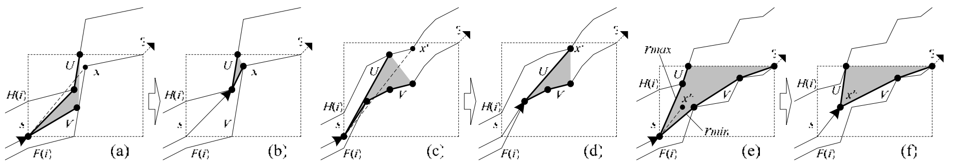

Figure 5: *ATS* generates the transmission schedule and reconstructs the funnel.

Unlike *SLWIN*(*k*), which independently computes the transmission schedule for each window, *ATS* remembers useful computational results for the next window by the help of a funnel data structure. Detail description of the funnel maintenance follows.

Appending the point x=(a, $F(a)$) onto V

As shown in Figure 6(a), along the lower chain *V*, *ATS* tries to find a point v_i ($1 \le i \le n$) from v_n to v_1 so that $Rate(v_{i-1}, v_i) > Rate(v_i, x)$. If such a point is found, *ATS* removes $v_{i+1}, \ldots, v_n$ from *V* and adds *x* to the tail of *V*. As shown in Figure 6(b), the resultant chain $V' = \{v_0, \ldots, v_i, x\}$ is concave. *V'* and *U* maintain the funnel. In this case, *ATS* does not generate the transmission schedule. If a point v_i is not found, along the upper chain *U*, as shown in Figure 6(c), *ATS* tries to find a point u_j ($0 \le j \le m-1$) from u_0 to u_{m-1} so that edge u_jx will not cross *U*, i.e. $Rate(u_j, x) < Rate(u_j, u_{j+1})$; otherwise, *ATS* sets $u_j = u_m$. As shown in Figure 6(d), *ATS* replaces *V* with $V' = \{u_j, x\}$. If j=0, *V'* and *U* maintain the funnel and *ATS* does not generate the transmission schedule. If $j \neq 0$, *ATS* generates the transmission schedule according to the chain $\{u_0, \ldots, u_j\}$ and then removes $u_0, \ldots, u_{j-1}$ from *U*. The resultant chain $U' = \{u_j, \ldots, u_m\}$ remains convex. *V'* and *U'* maintain the funnel again.

Appending the point $x' = (a, H(a))$ onto U

The processing of x' is similar to the processing of x. Thus, we skip this part.

Adding the point x'' at the head of V and U

As shown in Figure 7, along the lower chain *V*, *ATS* finds a point v_i ($0 \leq i \leq n-1$) from v_0 to v_{n-1} so that $Rate(x'', v_{i+1}) > Rate(v_i, v_{i+1})$ and thus $V' = \{x'', v_i, v_{i+1}, \ldots, v_n\}$ is concave. *ATS* removes $v_0, \ldots, v_{i-1}$ from *V* and adds x'' at the head of *V*. Along the upper chain *U*, *ATS* finds a point u_j ($0 \leq j \leq m-1$) from u_0 to u_{m-1} so that $Rate(x'', u_{j+1}) < Rate(u_j, u_{j+1})$ and thus $U' = \{x'', u_j, u_{j+1}, \ldots, u_m\}$ is convex. *ATS* removes $u_0, \ldots, u_{j-1}$ from *U* and adds x'' at the head of *U*. *U'* and *V'* maintain the funnel.

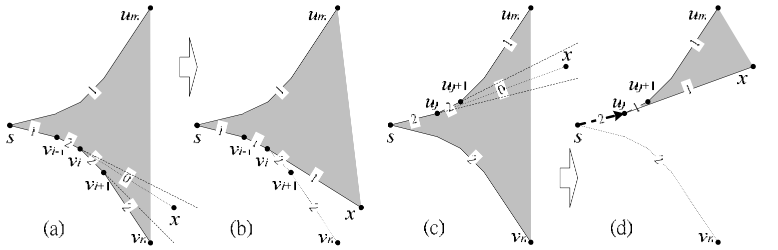
Figure 6: Processing x. The figure is skewed for better visualization.

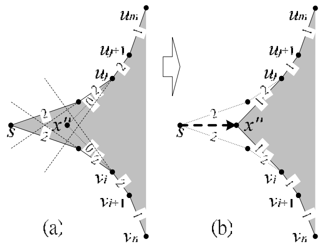
Figure 7: Processing x''. The figure is skewed for better visualization.

B. Complexity Analysis

We assume each edge that is added onto the funnel associates with a counter. The counter is initialized as zero and increases by one whenever the edge is scanned. The time complexity analysis of *ATS* is based on the following claims.

Claim A: The counter of an edge that is added onto the funnel and removed later is read as two.

Claim B: The counter of an edge that remains on the funnel is read as one.

The primary factor in the *ATS* time complexity is the maintenance of the funnel. Consider the process of appending x onto V. *ATS* first starts scanning the edges of the lower chain V from the tail. If there is a points v_i ($1 \le i \le n$) such that $Rate(v_{i-1}, v_i) > Rate(v_i, x)$, the scan stops. The edges of the chain $\{v_i, \ldots, v_n\}$ have been scanned and removed and their associated counters increased by one to two. The point x is then added to the tail of V. At this point, the readings of the two associated counters on the chain $\{v_{i-1}, v_i, x\}$ are two and zero. We can amortize the two counters so that each counter is reset to one, as shown in Figure 6(a) and (b). Thus, claims A and B hold. If there is no such point, *ATS* starts scanning the edges of the upper chain *U* from the head. Applying similar amortized analyses, we can prove that claims A and B also hold after the scan stops, as shown in Figure 6(c) and (d). Therefore, after *ATS* completes the process of appending x onto V, claims A and B hold. Similarly, we can prove that after *ATS* completes the process of appending x' onto U, claims A and B also hold. When there is no more frame size information available, *ATS* computes x'' in constant time and reconstructs the funnel, as shown in Figure 7. We can prove that after *ATS* completes the reconstruction, claims A and B hold.

ATS iteratively considers $F(i)$ and $H(i)$ once for $0 \le i \le N-1$. Each time, *ATS* adds one edge onto the funnel and may remove some edges. Since the number of transmission schedules generated will be N, at the most, *ATS* reconstructs the funnel at most N times. Each time, *ATS* adds two edges onto the funnel and may remove some edges. In total, *ATS* adds a maximum of $4*N$ edges onto the funnel. Thus, there is a maximum of $4*N$ associated counters and the summation of all readings is smaller than $8*N$. Therefore, the time complexity of *ATS* is $O(N)$.

IV. Experiment Results

In this section, we examine the performance of the proposed online traffic smoothing algorithm. The study focuses on network and client resources by measuring the peak rate and the buffer occupancy. The peak rate of a smoothed video stream determines its worst-case bandwidth requirement across the path from the video server to the client player. Hence, most traffic smoothing algorithms attempt to minimize the peak rate to increase the likelihood that the video server, network and client player have sufficient resources to handle the stream. Practically, media data may get lost in the network. However, if the client player can detect the data lost early enough, it can request a retransmission. A transmission schedule with a high percentage of client buffer occupancy usually implies that there is a high probability the client player will detect the data lost early.

We simulated the transmission of several MPEG video clips [18]. Table 1 shows some statistics of these clips. We compare the performance of *ATS* and *SLWIN*(1) (which consistently outperforms *SLWIN*(*W*)). Figure 8 and 9 show the peak transmission rate and buffer occupancy of the transmission schedules for transmitting Star Wars and News, respectively, when different initial delays and client buffer sizes are used. To demonstrate the lower bound of the peak rate, these figures also show the optimal offline schedules obtained by *MVBA* [1].

Table 1: Statistics of MPEG video clips. AVG is the average of frame sizes. STD is the standard deviation of frame sizes.

Stream	number of frames	FPS	GOP	maximum frame size	AVG	STD
Star Wars	40000	24	12	124816 bytes	9313.2	12902.73
News	40000	24	12	189888 bytes	15357.67	19505.57
MTV	40000	24	12	251408 bytes	19780.5	21453.2
TALK	40000	24	12	132752 bytes	17915	18222.2

A. Initial Delay

Given a client buffer size (*B*), when the initial delay (*D*) is small, *ATS* is more likely to use the aggressive workahead scheme to heuristically generate the transmission schedule in a window. When the initial delay increases, the possibility also increases that *ATS* deterministically generates the transmission schedule. As shown in Figure 8 and 9, the peak rates of the *ATS* transmission schedules are significantly smaller than *SLWIN*(1) when *D* is smaller than 30 frametimes. The aggressive workahead scheme reduces the suppression effect successfully. Since there is not much data, the buffer occupancies are around 30% to 50%. However, *SLWIN*(1) uses less than 20% of the client buffer. When *D* increases, the *ATS* transmission schedules may have the same peak rates as *SLWIN*(1). The buffer occupancies of the *ATS* transmission schedules keep increasing until to 80% or more and then slowly decreasing. They are consistently higher than *SLWIN*(1) even *D* is relatively large. When *D* increases beyond a certain point, *ATS* always deterministically generates the transmission schedule in every window. In such situation, *ATS* generates the same transmission schedule as *SLWIN*(1) and *MVBA*.

When *ATS* works ahead on Star Wars, an interesting phenomenon occurs, as shown in Figure 8(a) and (b). *ATS* dramatically reduces the peak rate to around 32 Kbytes per frametime when *D* is 12 frametimes. However, when *D* increases to 32 frametimes, *ATS* does not further reduce the peak rate. Analyzing Star Wars, we find there are two bursts. The second is slightly burstier than the first. If *D* is smaller than 32 frametimes, *ATS* cannot smooth the first burst. It therefore raises the peak rate. Since *ATS* aggressively works ahead and keeps less data in the video server, it can smooth the second burst. If *D* is larger than 32 frametimes, *ATS* is able to smooth the first burst. This time, however, *ATS* cannot smooth the second burst and therefore the peak rate increases.

B. Buffer Size

When the initial delay is small, enlarging the client buffer does not help *SLWIN*(1) reduce the peak rate. The suppression effect eliminates the benefits from a larger buffer. As shown in Figure 8 and 9, the peak rates of the *SLWIN*(1) transmission schedules further decrease only when *D* is larger than 120 frametimes and 80 frametimes, respectively. On the other hand, the aggressive workahead scheme uses the buffer more efficiently. The peak rates of the *ATS* transmission schedules are further reduced even *D* is small.

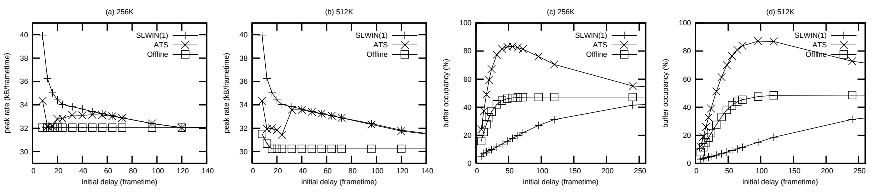

Figure 8: Comparisons of algorithms under different initial delays. (Star Wars, B=256K and 512K).

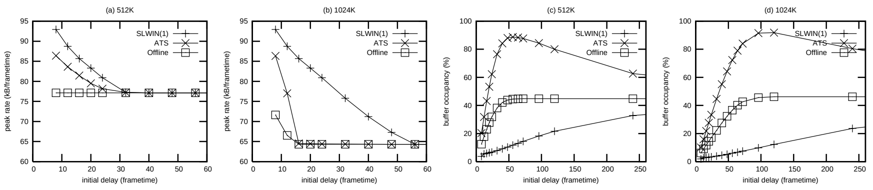

Figure 9: Comparisons of algorithms under different initial delays. (News, B=512K and 1024K).

V. Conclusion

In this paper, we present an efficient traffic smoothing algorithm *ATS* for live video transmission. Unlike *SLWIN*(*k*), which computes the smallest bandwidth requirement for each window independently, after *ATS* generates the transmission schedule for the current window, it remembers useful computational results for the next window by the help of a funnel data structure. The total time complexity of *ATS* is *O*(*N*). Note that *O*(*N*) is a trivial lower bound to online smooth an *N* frame live video. To reduce the suppression effect caused by the lake of future frame sizes, *ATS* heuristically uses the aggressive workahead scheme to generate the transmission schedule. We have evaluated *ATS* by transmitting several benchmark video clips. Experiment results show that *ATS* further reduces the peak bandwidth requirement and better utilizes the client buffer, especially for interactive applications in which the initial delay is small.

References

[1] James D. Salehi, Zhi-Li Zhang, Jim Kurose, Don Towsley, "Supporting Store Video: Reducing Rate Variability and End-to-End Resource Requirement Through Optimal Smoothing," *IEEE/ACM Trans. Networking*, Vol. 6, No.4, Aug. 1998.

[2] W. Feng and S. Sechrest, "Critical Bandwidth Allocation for Delivery of Compressed Video," *Computer Communications*, pp. 709-717, Oct. 1995.

[3] W. Feng, F. Jahaian and S. Sechrest, "Optimal Buffering for the Delivery of Compressed Video," *IS&T/SPIE MMCN*, pp. 234-242, 1995.

[4] R. I. Chang, M. Chen, M. T. Ko and J. M. Ho, "Optimization of stored VBR video transmission on CBR channel," *SPIE, VVDC*, pp. 382-392, 1997.

[5] R. I. Chang, M. Chen, M. T. Ko and J.M. Ho, "Designing the On-Off CBR Transmission Schedule For Jitter-Free VBR Media Playback in Real-Time Networks," *IEEE RTCSA*, pp. 1-9, 1997.

[6] J. M. McManus and K. W. Ross, "Video On Demand over ATM: Constant-Rate Transmission and Transport," *IEEE INFOCOM*, March 1996.

[7] J. M. McManus and K. W. Ross, "Dynamic Programming Methodology for Managing Prerecorded VBR Sources in Packet-Switched Networks," *SPIE VVDC*, 1997.

[8] M. Grossglauser, S. Keshav and D. Tse, "RCBA: A Simple and Efficient Service for Multiple Time-Scale Traffic," in *Proc. ACM SIGCOMM*, pp 219-230, Aug. 1995.

[9] Wuchi Feng, Jennifer Rexford, "A Comparison of Bandwidth Smoothing Techniques for the Transmission of Prerecorded Compressed Video," in *Proc. IEEE INFOCOM*, pp. 58-66, April 1999.

[10] J. Rexford, S. Sen, J. Dey, W. Feng, J. Kurose, J. Stankovic, and D. Towsley, "Online Smoothing of Live, Variable-Bit-Rate Video," in *Proc. International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pp. 249-257, May 1997.

[11] Subhabrata Sen, Jennifer L. Rexford, Jayanta K. Dey, James F. Kurose, Donald F. Towsley, "Online Smoothing of Variable-Bit-Rate Streaming Video," *IEEE Trans. Multimedia*, Vol. 2, No.1, March 2000.

[12] D. T. Lee and F. P. Preparata, "Euclidean Shortest Path in the Presence of Rectilinear Barriers," *Networks*, vol. 14, pp. 393-410, 1984.

[13] M. Garrett and W. Willinger, "Analysis, Modeling and Generation of Self-similar VBR Video Traffic," in *Proc, ACM SIGCOMM*, Sep. 1994.

[14] M. Krunz and S. K. Tripathi, "On the Characteristics of VBR MPEG Streams," in *Proc. ACM SIGMETRICS*, pp. 192-202, June 1997.

[15] A. Ads, "Supporting Real-Time VBR Video Using Dynamic Reservation Based on Linear Prediction," in *Proc. IEEE INFOCOM*, pp. 1476-1483, March 1996.

[16] S. Crosby, M. Huggard, I. Leslie, J. Lewis, B. McGurk and R. Russell, "Predicting Bandwidth Requirement of ATM and Ethernet Traffic," in *Proc. IEE UK Teletraffic Symposium*, March 1996.

[17] E. Amir, S. McCanne, and H. Zhang, "An Application Level Video Gateway," in *Proc. ACM Multimedia*, Nov. 1995.

[18] http://www3.informatik.uni-wuerzburg.de/MPEG/traces/

Algorithm ATS

```
tu=-D, tv=-D, t=-D;
Proc smooth() {
step 1:
    while (tu+1<tv) {
        if (H(tu+1)<=F(t+D) {
            tu=tu+1
            L=append_on_u(tu, H(tu));
            if (L>0) {return L}
        } else {
            goto step 3
        }
    }
step 2:
    while (tv+1<=t+D) {
        tv=tv+1
        L=append_on_v (tv, F(tv))
        if (L>0) {return L}
        if (H(tu+1)<=F(t+D)) {
            tu=tu+1
            L=append_on_u(tu, H(tu))
            if (L>0) {return L}
        } else {
            goto step 3
        }
    }
step 3:
    while (tv+1<=t+D) {
        tv=tv+1
```

```
            L=append_on_v(tv, F(tv))
            if (L>0) {return L}
        }
step 4:
        y2=G(t)+MAX(MIN(Rmax, Rheap), Rmin)
        L=Generates (t, G(t))->(t+1, G(t)+y2))
        reconstruct the funnel
        return L
}
Proc Main(V, N) {
        while (t<N) {
            L=smooth()
            t=t+L
        }
}
```