*Submission Number: C-144*

# Exploiting Temporal Parallelism For Software-only Video Effects Processing

Ketan Mayer-Patel Lawrence A. Rowe
{kpatel,rowe}@cs.berkeley.edu

## Abstract

Internet video is emerging as an important multimedia application area. Although development and use of video applications is increasing, the ability to manipulate and process video is missing within this application area. Current video effects processing solutions are not well matched for the Internet video environment. A software-only solution, however, provides enough flexibility to match the constraints and needs of a particular video application. The key to a software solution is exploiting parallelism. This paper presents the design of a parallel software-only video effects processing system. Preliminary experimental results exploring the use of temporal parallelism are presented.

## 1 Introduction

Internet packet video is emerging as an important multimedia application area. The Multicast Backbone (MBone) conferencing tool *vic* and *NetMeeting* from Microsoft are examples of Internet packet video conferencing tools. *RealVideo* from RealNetworks, *NetShow* from Microsoft, and *IP/TV* from Precept are examples of video-on-demand streaming Internet packet video applications.

These applications are used for a variety of purposes and audiences. Using a video-on-demand system, a single recipient views stored material. Video conferencing is used by small groups for meetings and by larger groups to attend remote events (e.g., classes, conferences, concerts, etc.). Large broadcasts involve thousands of people "tuned" into an on-going program originating from either live sources or stored archives. For this paper, we group all of these applications under the general term *Internet Video* (IV).

Internet video is characterized by variability. Frame rates, bit rates, and jitter are all variable in IV applications. In contrast, a television system, whether analog or digital, is characterized by constant frame rates, bounded bit rates, and synchronous transmission. IV applications must also deal with packet loss. These applications often use multicast networking protocols.

Although development and use of IV is increasing, the ability to manipulate and process video is missing. Live broadcasts of conferences, classes, and other special events require improved production values. Experience from the television, video, and film industries shows that visual effects are an important tool for communicating and maintaining audience interest [9]. Titling, for example, is used to identify speakers and topics in a video presentation. Compositing effects that combine two or more video images into one image can be used to present simultaneous views of people or events at different locations or artifacts at varying levels of detail. Blends, fades, and wipes are transition effects that ease viewers from one video source to another. Figure 1 illustrates several video effects.

Traditionally, video effects are created using a video production switcher (VPS). A VPS is a specialized hardware device that manipulates analog or digital video signals to create video effects. It is usually operated by a technician or director at a VPS control console. Figure 2 shows a Composium VPS produced by DF/X.

A conventional VPS is not well matched for the IV environment. An analog VPS requires signals with very tight timing constraints which are not present with video on the Internet. A digital VPS requires uncompressed signals and uses networking protocols not suitable for the Internet. Moreover, hardware VPS solutions can be very expensive. A VPS can cost anywhere from $1000 for a low-end model with very limited capabilities to $250,000 for a full featured digital VPS like the Composium pictured in Figure 2.

Another disadvantage of conventional hardware solutions is the difficulty incorporating them into a networked environment. One advantage of IV applications is the ability for users to participate from their desktop. The cost of video processing hardware makes it impractical to provide each user with his or her own


Titling

Transition Effect

Compositing

Figure 1: Sample Video Effects

VPS. Some organizations build a traditional video routing network alongside the computer network to connect desktops with video processing hardware located in a machine room. This approach is also impractical.

We are developing a software-only video effects processing system designed for IV applications. A software-only solution using commodity hardware provides flexibility that matches the constraints and needs of these applications. Software systems can be written to handle video formats already in use on the Internet and to use multicast communication protocols.

Different IV applications require different tradeoffs between latency, bandwidth, and quality. Interactive conferences may demand low latency at the expensive of quality and bandwidth. Broadcast presentations can tolerate higher latencies to achieve higher quality. The number of viewers and type of content also impact the desired quality. Conventional television systems cannot vary these quality and delivery parameters.

The key to a software solution is exploiting parallelism. Currently, a single processor cannot produce a wide variety of video effects in real-time which is why conventional VPS systems and early research systems (e.g., Cheops [1]) use custom-designed hardware. Even as processors become faster, the demand for more complicated effects, larger images, and higher quality will increase. The complexity of video effects processing is arbitrary because the number, size, data rate, and quality of video streams is variable. Unlike CD quality audio, which is near the limits of human perception, the quality of video used on the Internet is quite poor. Improvements in processor and networking technology will only be met with greater application demands.

Fortunately, video processing contains a high degree of parallelism. Three types of parallelism can be exploited for video effects processing: functional, temporal, and spatial. Functional parallelism decomposes the video effect task into smaller subtasks and maps these subtasks onto the available computational resources. Temporal parallelism can be exploited by demultiplexing the stream of video frames to different processors and multiplexing the processed output. For example, one processor may deal with all odd numbered frames while another deals with all even numbered frames. Spatial parallelism can be exploited by assigning regions of the video stream to different processors. For example, one processor may process the left half of all video frames while another deals with the right half.


Figure 2: A Video Production Switcher

Taking advantage of these types of parallelism requires the solution of different problems. Exploiting functional parallelism requires the application of compilation techniques to produce an efficient decomposition of the processing task into smaller components. Temporal and spatial parallelism require mechanisms

for distributing input video streams to the appropriate processor and recombining the resulting output.

This paper describes the design of a parallel software-only video effects processing system. A general software architecture is presented that will run on a set of computers connected by a high bandwidth, low latency network. Specific mechanisms for exploiting temporal parallelism have been implemented and are described. These mechanisms address the problem of distributing input frames among different processors and recombining the output to produce a single video stream. Performance measurements for these mechanisms are presented and evaluated.

The remainder of the paper is organized as follows. Section 2 discusses related work. The system architecture is described in Section 3. Details of the specific mechanisms required for temporal parallelism are given in Section 4. Results from experiments measuring the performance of these mechanisms are presented in Section 5. Section 6 concludes the paper and outlines the direction of future work.

# 2 Related Work

Several hardware systems have been developed to explore parallel video effects processing. The Cheops system developed by Bove and Watlington at MIT is composed of interconnected special-purpose hardware components that implement specific functions (e.g., discrete cosine transform (DCT), convolution, etc.) [1]. Video effects are implemented by configuring and controlling data flow between these specialized hardware components. This system focused primarily on exploiting temporal and functional parallelism. The IBM Power Visualization System is a parallel processor composed of up to 32 identical processors interconnected by a global bus [4]. It was designed specifically to support the IBM EFX suite of editing, effects, and compression software. The Princeton Engine is a parallel processor composed of up to 2048 custom-designed processing elements which are used to simultaneously operate on an array of data elements (i.e., SIMD) [2]. Many other hardware systems have also been developed [6, 7, 11]. The system proposed here differs fundamentally from these systems by not assuming any particular underlying parallel architecture.

More recent work by Bove and Watlington describes a general system for abstractly describing media streams and processing algorithms that can be mapped to a set of networked hardware resources [16]. In this system, hardware resources may be special-purpose media processors or general-purpose processors. The system is centered around an abstraction for media streams that describes any multi-dimensional array of data elements. The system achieves parallelism by discovering overlaps in access patterns and scheduling subtasks and data movement among processors to exploit them. The system uses a general approach that is not specific to video or packet video formats and that is independent of networking protocols. This research shares some of the same goals and solutions that we are working toward. Our system is different in that we are taking advantage of representational structure present in compressed video formats, and we are constrained to standard streaming protocols for video on the Internet (i.e., RTP [12]).

The Resolution Independent Video Language (RIVL) is a high-level language for describing video effects irrespective of format and resolution [15]. The system described here is independent of any specific language for specifying video effects, but RIVL serves as a model for the type of language to be supported. Dali is a low-level set of image operators that operate on specific representations of data elements [13]. We are using Dali as a target language to express primitive effects processing tasks.

# 3 System Architecture

The overall system architecture is shown in Figure 3. This picture depicts the high bandwidth, low latency network as a cloud. The oval labeled "IV Application" represents an application that requires video effects processing. The application sends a specification of the desired video effect to the "Effects Server." This specification also identifies the video input streams which may be live or stored video. The ovals labeled "Effects Processor" represent general-purpose computers. The "Effects Server" allocates system resources, maps the effect onto the available processors, and provides the IV application with a means to control the effect (e.g., change any relevant parameters).

## 3.1 Design Goals

This subsection outlines the major design goals for the proposed system. Four primary goals that influenced the system design are:

1. Exploit all available means of parallelism.
2. Use standard video formats and networking protocols.

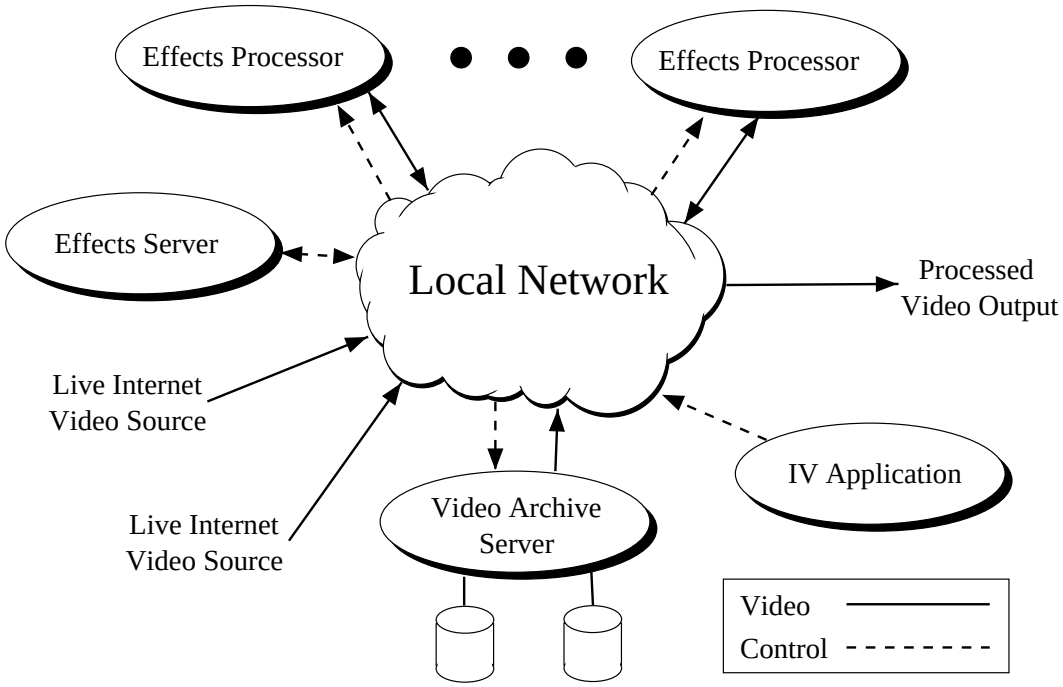


Figure 3: System Architecture

3. Use commodity hardware.

4. Service multiple effect-tasks simultaneously.

The system should exploit all types of parallelism: functional, temporal, and spatial. To support functional parallelism, the system must compile a high-level description of a video effect into an interrelated set of low-level subtasks which are appropriate for the computational resources (i.e., number of available processors). Temporal and spatial parallelism require mechanisms to distribute and reassemble video data to and from processors. Moreover, the system should be able to use different types of parallelism in conjunction with each other to achieve an appropriate implementation of the desired effect. One challenge for the system will be to decide which types of parallelism are appropriate for a particular video effect. Format and operation specific cost models will be developed to predict performance so that good solutions can be constructed.

The second major design goal is to work within standard video formats and networking protocols found on the Internet. Motion-JPEG, H.261/H.263, and MPEG are the most common video formats in use today. These formats are very similar – they use the DCT, quantization, and entropy coding with intra- and inter-frame optimizations. We will capitalize on available compressed domain processing methods wherever possible [14]. RTP is the standard networking protocol used by IV applications [12]. This protocol is used for input and output video streams.

The third major design goal is to use commodity hardware. The system should operate on any set of networked, general-purpose processors. This goal requires the system to be software-only and portable. Clearly, system performance will be limited by communication latency and processing speeds of the particular resources available. The most common architecture will be 2 to 100 processors connected by a fast, low latency local area network. The Network of Workstations (NOW) project at U.C. Berkeley provides this type of environment and is the target for our development efforts [3].

The last design goal is to support multiple effect-tasks simultaneously. Several IV applications may require effects processing from the system at the same time, or one application may instantiate two or more effects at the same time. The system processing resources must be dynamically allocated.

This paper concentrates on the software components needed to map a particular video effect onto a given set of resources and the overhead of temporal parallelism.

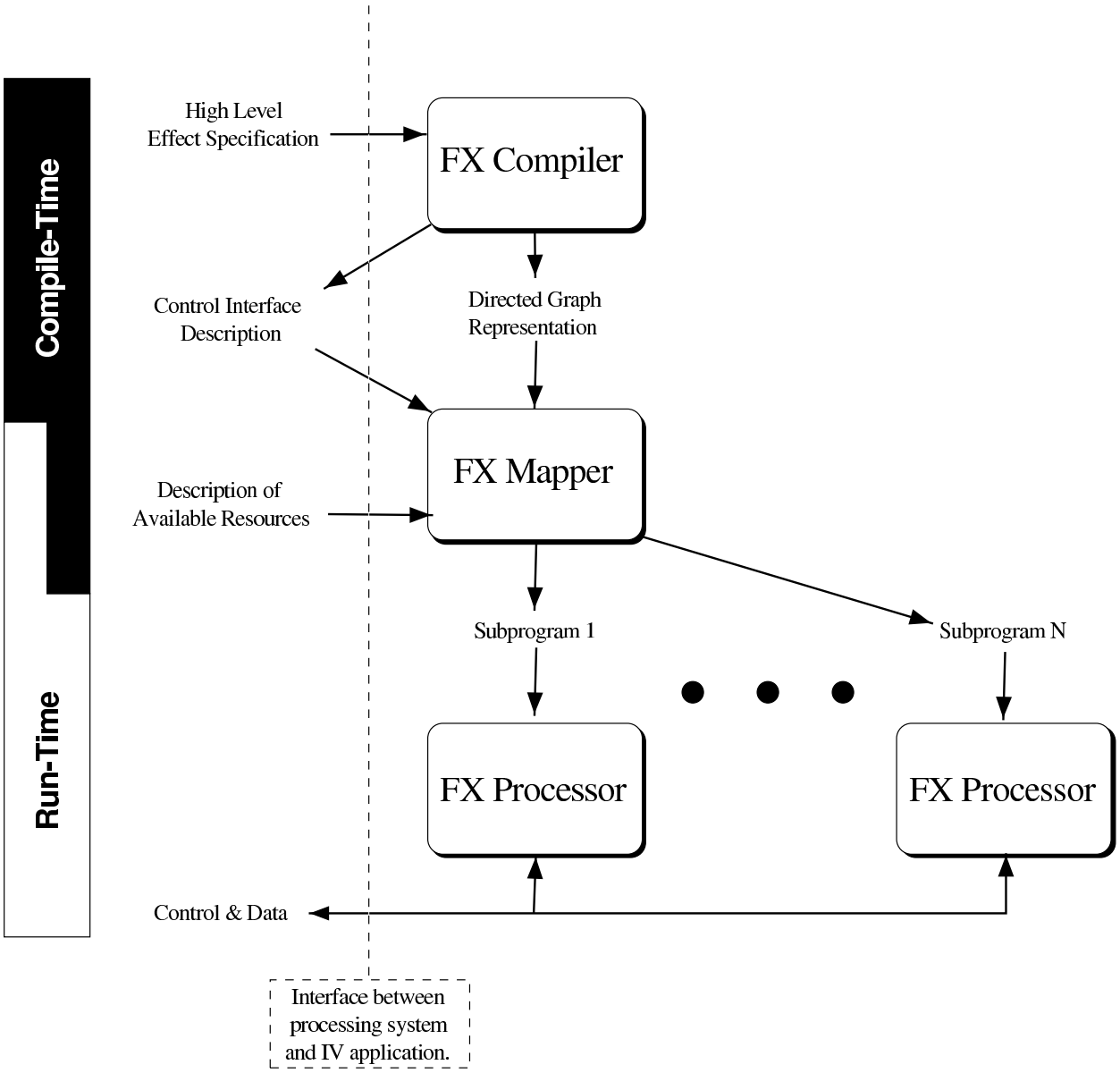


Figure 4: Software Architecture

### 3.2 Software Architecture

The system is composed of three major software components: the FX Compiler, the FX Mapper, and the FX Processor. The relationship between these components is illustrated in Figure 4. The FX Compiler translates a high-level description of a video effect into an intermediate representation suitable to exploit functional parallelism. The FX Mapper takes the intermediate representation and maps it onto the available resources. The FX Mapper produces effect "subprograms" that will be executed on a particular computational resource. The FX Processor executes these subprograms and responds to control signals sent from the application. Placing these components in the overall system architecture depicted in Figure 3, the FX Compiler and FX Mapper are part of the "Effects Server" and the FX Processor is the software executing on an "Effects Processor."

Figure 4 shows the FX Compiler as a compile-time component, the FX Processor as a run-time component, and the FX Mapper as both a compile- and run-time component. Compile-time refers to parts of the system that do not depend on knowing exactly how many processors are available or specifically which video streams will be inputs. Run-time refers to components that are used when an effect task is executed.

The FX Compiler is the bridge between high-level effect descriptions in a language like RIVL and an intermediate form appropriate for mapping onto parallel computation resources. Our strategy for developing the FX Compiler is to use a directed graph of video operators as the intermediate form (i.e., a data flow graph). The video operators are the "instruction set" available to the FX Compiler.

For example, consider the cross-dissolve video effect shown in Figure 5. A directed graph representation of this effect is shown in Figure 6. The directed graph is made up of three nodes. Each node represents a video operator. Two of the nodes represent the operation of multiplying each pixel by a scalar value. The third node represents the function of adding two frames together. The cross dissolve is implemented by varying the parameter $p$ from 0.0 to 1.0.

The granularity of the operators will determine how much functional parallelism can be exploited. The price for fine grained operators, however, will be increased overhead. The trade-off between granularity and overhead is a research issue that will be explored.

Another responsibility for the FX Compiler is constructing a control interface description for the graph. In the cross dissolve example, the parameter $p$ deter-

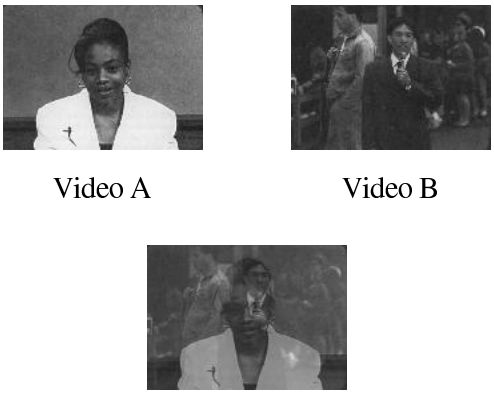


Figure 5: Cross-Dissolve

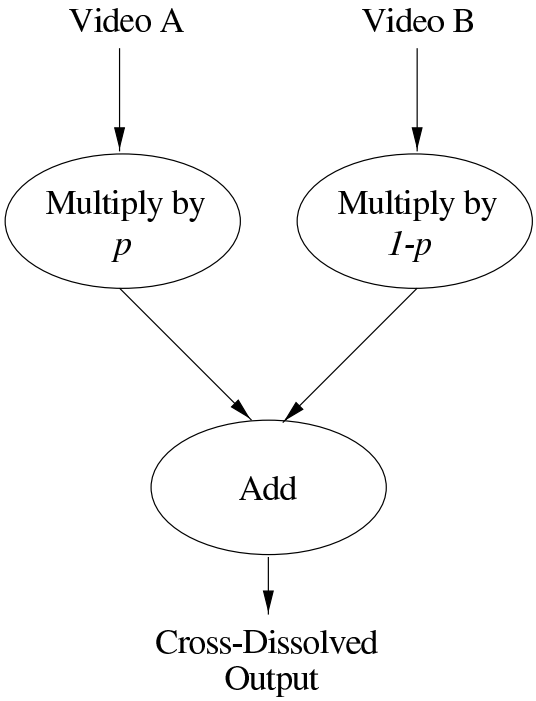


Figure 6: Cross-Dissolve Directed Graph Representation

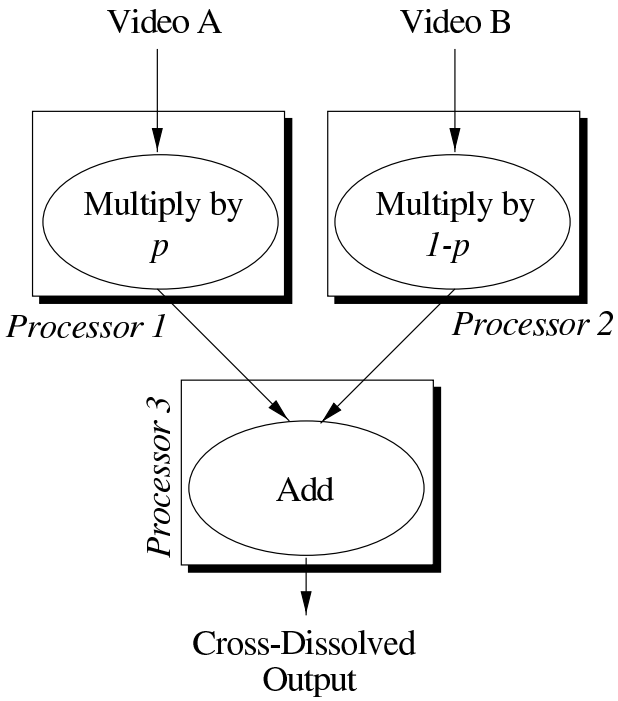


Figure 7: Functional Partition of Cross-Dissolve Graph

mines to what degree one video source is dissolved into the other. A more complicated effect may have many parameters that are interrelated. A control interface description will name each parameter, indicate legal values for the parameter, and possibly relate the parameters to each other. Applications using the system will use the control interface description to manage the effect by generating an appropriate user interface, mapping the controls to a predefined interface (e.g., a software-only video production switcher interface [17]), or controlling the effect programmatically[1].

The FX Mapper is analogous to a database query optimizer or a compiler code generator. The FX Mapper determines how the video effect will be parallelized. The effect graph produced by the FX Compiler is augmented with temporal and spatial parallelism operators. The graph is then partitioned into subgraphs which are mapped to computational resources. Again consider the cross dissolve example illustrated in Figure 6. Figure 7 shows a possible partitioning of this graph using functional parallelism.

[1]For example, building a heuristic program that uses all sources of input (e.g., media streams, remote control commands, user preferences, etc.) to automate the production of a video program typically produced by a human director.

In this example, each video operator is mapped to a different processor. Figure 8 shows a possible partitioning using temporal parallelism. In this example, the graph is augmented with operators for controlling the temporal subdivision and interleaving. Figure 9 shows the same example using spatial parallelism.

Our approach to building the FX Mapper is to use a recursive bipartitioning process to generate possible configurations. A predictive cost model for the video operators will be developed to estimate the performance of these configurations and choose the best one. The bipartitioning process starts by choosing a particular type of parallelism to exploit and dividing the graph into two subgraphs. Recursively, each subgraph is processed in the same manner until the number of subgraphs equals the number processors available. Heuristics incorporating the predictive cost model will be used to guide the partitioning process. Different partitions generated at each step produce the sequence of plans to be evaluated by the FX Mapper. Once a plan is selected, the FX Mapper generates subprograms that implement the effect. The effect graph primitives are implemented as Dali programs or operations implemented in a general purpose programming language.

The FX Processor is the execution agent for the subprograms generated by the FX Mapper. The execution environment is implemented on the MASH platform [8]. MASH is a flexible software environment for building distributed continuous media applications. It supports existing Internet protocols including RTP, RTCP, RTSP and SRM [12, 5, 10].

# 4 Temporal Parallelism

Exploring the use of temporal parallelism is a logical starting point for this project for several reasons. First, the processing subgraph for each computational resource is the same. In fact, we can abstract away the effect specification issues which will be handled by the FX Compiler and Mapper and treat the subgraph as a black box. Performance of mechanisms to support temporal parallelism can be measured independent of the specific processing task.

Two functions must be provided to support temporal parallelism: select frames to send to a particular processor and interleave the resulting output streams. We call these the *selector* and *interleaver* functions, respectively.

RTP, the network communication protocol, directly influences the solutions to these two issues. Each RTP payload type (i.e., video format) uses a distinct packet

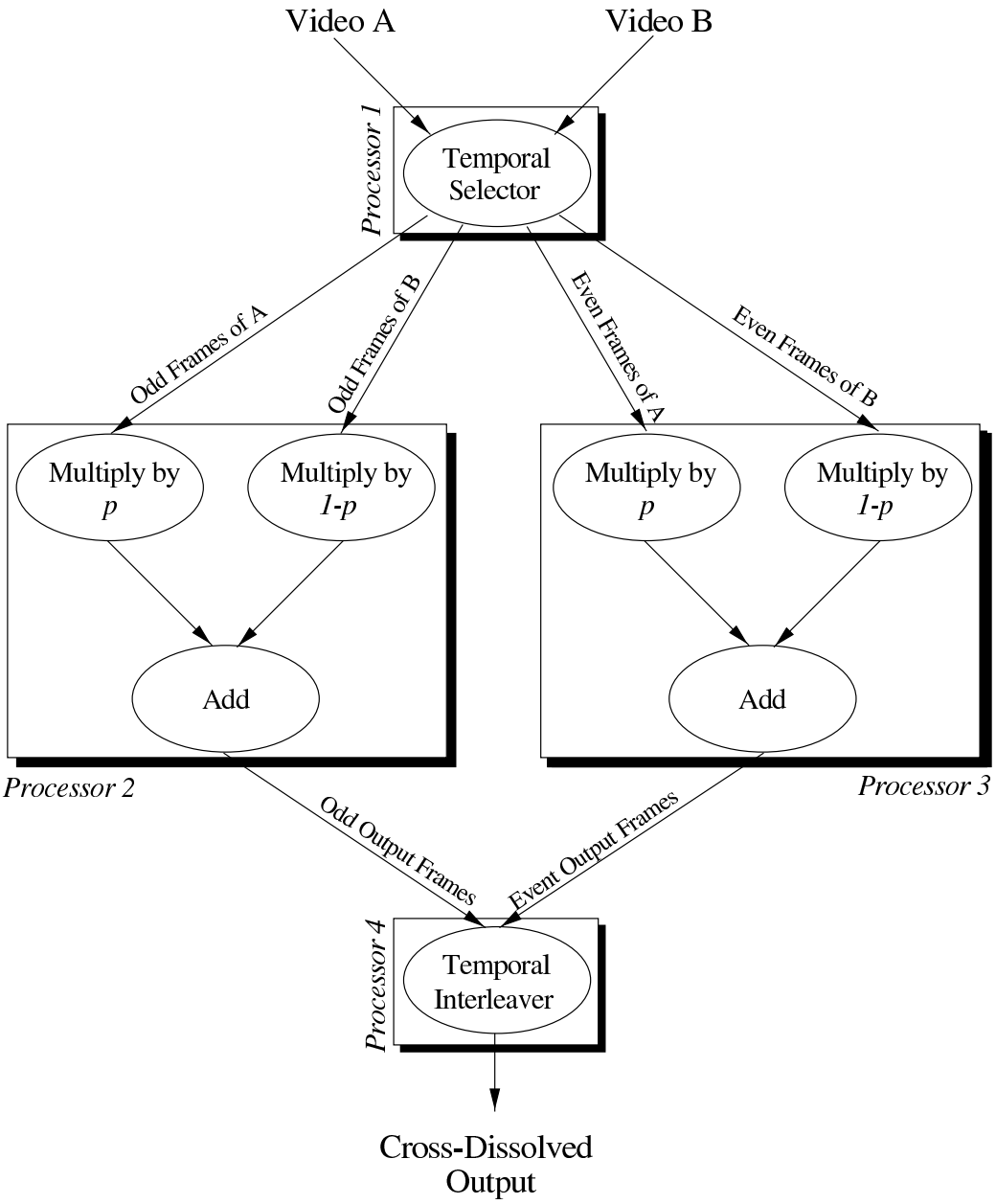


Figure 8: Temporal Partition of Cross-Dissolve Graph

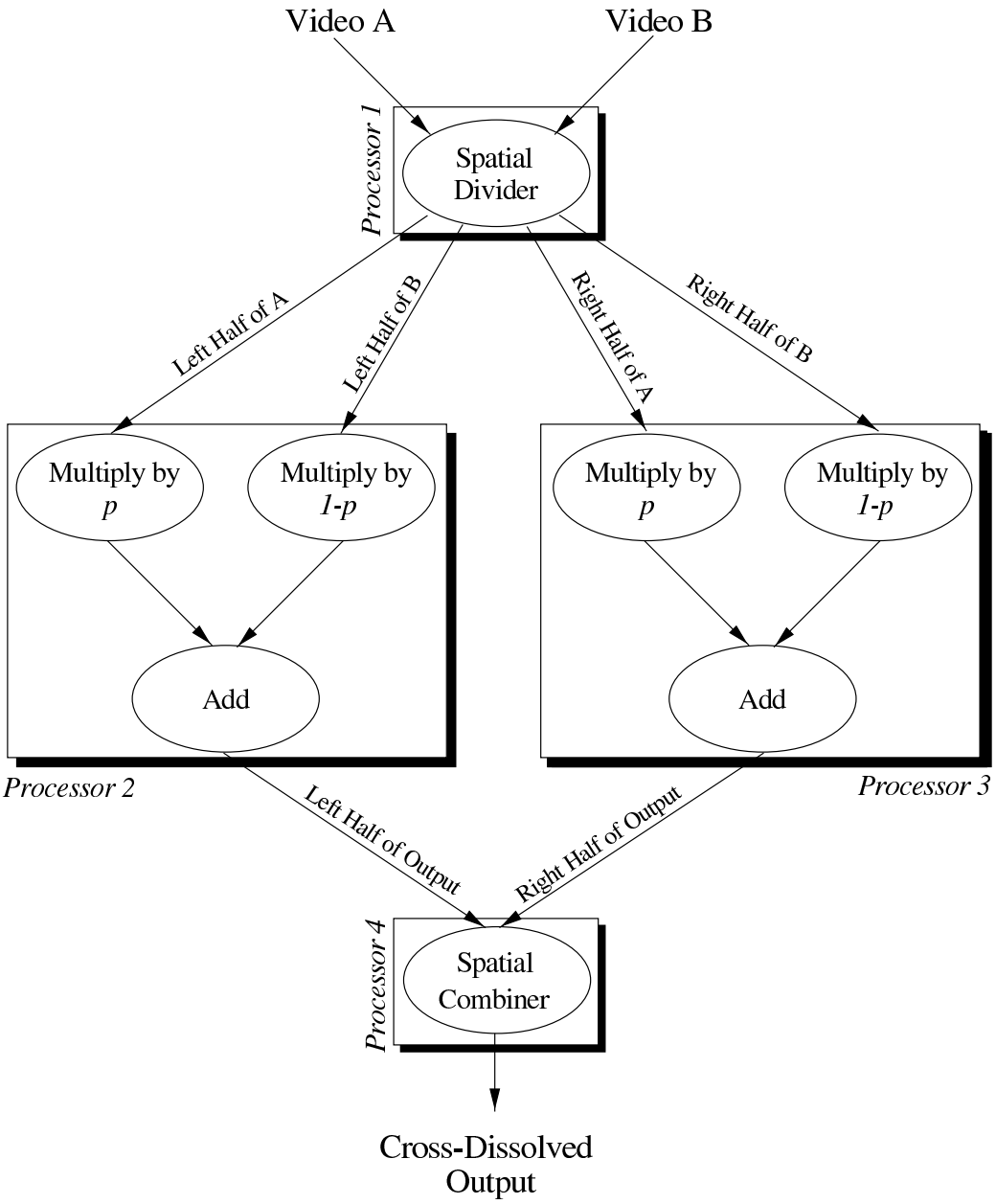


Figure 9: Spatial Partition of Cross-Dissolve Graph

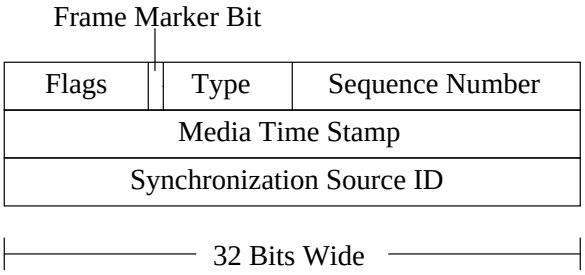


Figure 10: RTP Header

format and fragmentation scheme. In the initial experiments described below, we assume an MJPEG payload since frames in this format do not have temporal dependencies. Common to all RTP packets, however, are four key pieces of information: the synchronization source id (SSRC); the media timestamp (MTS); the packet sequence number (PSN); and the frame marker bit (FMB). Figure 10 shows where this information is stored in an RTP header. The selector and interleaver functions work with only these fields.

The MTS is sampled from a payload specific clock, which for most video sources runs at 90kHz. For the MJPEG payload type, each frame is fragmented into one or more packets because a single frame, which is typically 1 to 8kB, does not fit into a packet which is typically 1kB. All packets for a particular frame have the same MTS. Note that the MTS is not a frame number because the media clock runs at a faster rate than the video stream frame rate. The difference between the MTS values of two frames provides a measure of instantaneous frame rate. The PSN is unique to a particular RTP packet. If packets are delivered in order, PSN values will increase monotonically. The FMB has a payload specific meaning. For MJPEG, the FMB indicates the last packet for the frame associated with a specific MTS. The SSRC uniquely identifies the sending process.

## 4.1 Selector Function

We explored two possible strategies for selecting frames for processing agents: a decentralized approach in which each processor reads all frames but only processes its appropriate share, and a centralized approach in which a specific agent is responsible for forwarding packets to the appropriate processor.

On the surface, the decentralized approach is attractive because it avoids a central bottleneck and allows for a higher degree of scalability. Also, an extra latency penalty is avoided because frames can be multicast to the FX Processors.

The disadvantages of the decentralized approach, however, are quite severe. First, a decentralized algorithm for deciding whether or not a particular frame should be processed by a processor given only the MTS and PSN is difficult to construct. The algorithm cannot assume that the difference between any two consecutive MTS values is constant since the frame rate may change. Even if we do not expect the frame rate to change often, small variations in timing are common. Another approach might be based on counting frames, processing every $n$th frame where $n$ is the number of processors. Although there is a small start-up cost for making sure that no two processors operate on the same set of frames, this approach works well in the absence of lost packets. Unfortunately, when the frame rate of the input stream exceeds the performance of a given processor, packets are lost and the scheme breaks down. Control information must be exchanged between processors in order for any decentralized scheme to be stable which adds to algorithm complexity.

Second, synchronization among input streams is difficult. If the effect involves more than one input stream (e.g., compositing or transitions), the selector function must not only identify the correct frames from one sequence, but also select the appropriately synchronized frame(s) from the other input stream(s). This task is complicated further by the fact that the input streams may be arriving at different frame rates.

Third, the speed-up achieved by the decentralized approach is dominated by the time required to read frames that are not processed. For example, if 100 processors are processing a particular input stream using temporal parallelism, each processor will read 100 frames of data from the network to produce the 1 frame for which it is responsible. The time spent reading the other 99 frames is wasted.

The centralized approach, on the other hand, provides a simpler solution at the expense of a latency penalty. Frame rate changes and packet drops can be handled consistently and without interprocess communication. Synchronization remains a problem, but at least information about the assignment of frames to processors is known and coordinated by the selector function.

The disadvantage to the centralized approach is the overhead of reading a frame from the network, deciding which processor to send it to, and then writing the frame back to the network. Also, the speed of the selector function determines the maximum frame rate that can be achieved.

We want to use the centralized approach for its simplicity and to avoid the severe disadvantages and complexity of the decentralized approach. The problem is the throughput limit imposed by central control and the latency of passing through another process. To study the magnitudes of these problems, we built a simple selector module that distributes frames to participating processors in a round-robin manner. Experiments that measure throughput and latency are presented in Section 5.

## 4.2 Interleaver Function

The interleaver function must be centralized because of constraints imposed by RTP. Since packets of the interleaved output stream share the same SSRC (see Figure 10), packets that originate from different source addresses (i.e., processors) will appear as an SSRC conflict to applications that receive the merged output stream. Even if the SSRC conflict was resolved, PSN's must increase monotonically which requires each processor to communicate the number of packets used for each frame accurately and quickly. Finally, given variable processing latencies for each frame, the interleaver must adaptively buffer packets and reorder them. A decentralized approach to this problem suffers from the same disadvantages as the decentralized selector function.

The interleaving problem is similar to the problem of smoothly displaying video frames solved by video playback applications. The interleaving problem is different in that the variability of frame arrival is likely to be more severe and smoothness is not the primary goal. The primary goal when constructing the interleaver is to minimize buffering latency while avoiding frame drops. In the simplest and best case when frames arrive from the processors in order, packets can be forwarded by the interleaver without any buffering delay. If frames arrive out of order, the interleaver must buffer packets for reordering. A frame that arrives after a subsequent frame has already been forwarded must be dropped to preserve the RTP semantics of the MTS for the MJPEG payload type. The challenge is to dynamically adjust the buffering required to avoid frame drops while minimizing buffering latency.

Like the smooth playback problem, the key to solving the interleaving problem is to construct a mapping between the MTS of arriving frames and a local clock to schedule frame transmission. This mapping incorporates the idea of a *playout delay*. The playout delay is the delay incurred (i.e., the expected latency) if the frame arrives as expected. Thus, a frame can be delayed up to this amount and still be properly transmitted. If a frame arrives earlier than its scheduled playout time, it is buffered.

Our design for an interleaver function uses late frames to signal when the playout delay should be increased and frames that arrive in order to signal when the playout delay should be decreased. A tunable parameter governs how aggressively the interleaver reacts to either situation. How these adjustments are made is described next.

The following definitions are used to describe the interleaver function:

$M(f)$ : MTS of frame $f$.

$T(f)$ : arrival time of frame $f$. This time is expressed in the same units as $M(f)$ (i.e., sampled from a 90kHz clock for RTP video streams).

*offset* : mapping between MTS and local time (i.e., playout delay).

$\Delta_{est}$ : estimated MTS difference between consecutive frames.

$Q$ : priority queue of frames waiting for transmission ordered by their MTS.

$h$ : next frame to be sent in $Q$ (i.e., head of $Q$).

$\alpha$ : latency bias parameter ranged in $[0, 1]$.

$\beta$ : frame skip tolerance parameter ($> 0$).

$l$ : last frame transmitted.

The value $M(f) + \mathit{offset}$ is the scheduled time for transmitting $f$. In the ideal case, when all frames arrive in order and equally spaced (i.e., no jitter), *offset* is set so that $M(f) + \mathit{offset} = T(f)$. In other words, *offset* is set so that frames are scheduled to be transmitted at the same time as they arrive at the interleaver. The first frame to arrive at the interleaver is used to set *offset* so it is transmitted immediately. For all subsequent frames the following algorithm is used to adjust the playout delay *offset*:

```
   recv(f)
1     /* Function that receives frame f */
2     if (M(f) < M(l))
3        /* Frame is late. We must drop it. */
4        /* Adjust offset to increase buffering. */
5        offset = α * offset + (1 − α) * (T(f) − M(f))
6     else if (M(f) + offset < T(f))
7        /* Frame is late, but still valid */
8        transmit(f)
```

```
9     else if (M(f) − M(l) < β ∗ Δ_est)
10        /* Frame is early, but is probably */
11        /* the next expected frame. */
12        transmit(f)
13        offset = (1 − α) ∗ offset + α ∗ (T(f) − M(f))
14    else
15        /* Frame is early so add to Q */
16        queue-insert(f)
17
18    if (h ≠ f)
19        /* Try to process head of queue if */
20        /* different from this frame. */
21        g =remove-queue-head()
22        recv(g)
23    set-timer(h)
  end recv
```

The nested conditional statements on lines 2, 6, 9, and 14 classify a frame into the following categories:

**Late** The frame is too late to be transmitted (line 2).

**Late but valid** The frame is late (i.e., its scheduled transmission time has already passed), but can still be transmitted since no subsequent frame has yet been sent (line 6).

**Early but expected** The frame is early (i.e., its scheduled transmission time is in the future), but the difference between its MTS and the MTS of the last transmitted frame is within some tolerance of our estimate of the current frame rate (line 9).

**Early** The frame is early and does not appear to be the next frame we expect to transmit (line 14).

If the frame is late, the value of *offset* is adjusted to increase buffering to avoid future frame drops. The value $\alpha$ determines how much adjustment is made. A value of $\alpha$ near 0 makes large adjustments and a value of $\alpha$ near 1 makes small adjustments. If the frame is late but valid, the frame is immediately transmitted and no adjustment is made to *offset*.

If the frame is early, the parameters $\beta$ and $\Delta_{est}$ determine whether or not the frame is the next expected frame. Although not shown in the pseudo-code, the value of $\Delta_{est}$ is a moving average estimate of the MTS difference of consecutive frames. This estimate is updated every time a frame is transmitted. The value of $\beta$ determines how sensitive the algorithm is to changes in frame rate. If the frame is the next expected frame, it is sent immediately and *offset* is adjusted to reducing buffering. In this case, a value of $\alpha$ near 0 makes small adjustments and a value of $\alpha$ near 1 makes large adjustments. If the frame is too early, the frame is inserted into the queue. The **set-timer** function on line 23 schedules transmission for the current head of the queue.

Every time a frame is processed, the head of the queue is reprocessed to determine if it can be successfully sent (lines 18-22). This check is especially important if the frame at the head of the queue is the next expected frame. Sending the next expected frame early is how the algorithm makes adjustments to reduce buffering latency.

The value chosen for $\alpha$ determines the trade-off between reducing latency and avoiding frame drops. Consider the two extreme cases. When $\alpha = 0$, an adjustment to *offset* is only made when a late frame is encountered. The adjustment will ensure that any future frame that is delayed by up to the same amount of time will not be dropped. In this case the playout delay is set to accommodate the longest delay thus far seen. When $\alpha = 1$, no adjustment is made for late frames. The value of *offset* is set to send the next expected frame as soon as possible. In this case, very little buffering is done to avoid frame drops.

We have built an interleaver that uses this algorithm. Results from an experiment that demonstrates the tradeoff between latency and buffering are given in the next section.

# 5 Experimental Results

The results of two different experiments are presented in this section. The first experiment measures the performance of the centralized temporal selector as the number and frame rate of input streams is increased. The second experiment measures the buffering latency and frame drop percentage of the temporal interleaver for different values of the parameter $\alpha$ described above.

Both experiments were conducted on the U.C. Berkeley Sparc NOW. The Sparc NOW is composed of 100 Sun Ultra-Sparc1 computers connected by a Myrinet switch running at 160 MB/s.

## 5.1 Selector Performance

To measure selector performance, we multicast 1Mb/s MJPEG RTP video streams with a target frame rate of 30 fps and used the temporal selector to distribute the frames to four FX Processors. These four FX Processors received both the selected frames from the

| Number Of Streams | Latency (ms) | Std. Dev. (ms) | Frame Drop Percentage |
|---|---|---|---|
| 1 | 4.9 | 17.1 | 0.0 |
| 2 | 13.3 | 35.8 | 0.0 |
| 3 | 36.3 | 70.5 | 0.0 |
| 4 | 90.5 | 97.3 | 1.7 |
| 5 | 151.9 | 92.3 | 3.8 |

Table 1: Selector Performance

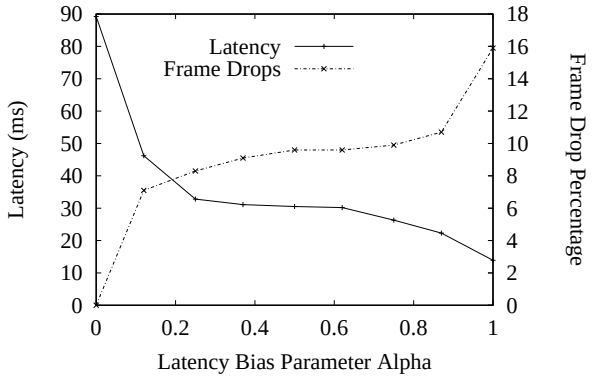


Figure 11: Latency and frame drops as a function of $\alpha$.

temporal selector as well as the frames from the original multicast streams. By recording the arrival times of all frames, the FX Processors can measure the latency incurred by the temporal selector. The selector was also responsible for synchronizing streams when more than one input stream was present. Because our approach to building the selector is independent of the number of processors involved (i.e., there is little per processor state and no part of the algorithm is dependent on the number of processors involved), the fact that we used four processors does not affect selector performance. Table 5.1 shows the results of the experiment.

As expected, adding more input streams increased selector latency. The standard deviation indicates that the selector also introduced jitter. For example, the standard deviation in the two-input case indicates that latency values often varied by as much as one frame time (i.e., 1/30 of a second). The percentage of frames dropped by the selector is low even for five input streams.

## 5.2 Interleaver Performance

The interleaver performance was measured by streaming video through four FX Processors which introduced a random processing delay chosen from an exponential distribution. The frames were distributed to the four FX Processors by the selector function. This design allows us to simulate a video effects processing task which caused frames to arrive at the interleaver out of order. The delay introduced was on average 33 ms. The latency bias parameter (i.e., $\alpha$ in the above description) was varied from 0 to 1. Latency (i.e., amount of time in the interleaver buffer) and frame drop percentages were recorded. Figure 11 shows the average latency and frame drop percentage as a function of the latency bias parameter $\alpha$. Although the latency and drop rate are related to $\alpha$ as expected, the parameter does not provide a smooth tradeoff. A traditional video industry standard is to keep latency through processing components under one frame time. For our temporal interleaver, this goal can only be achieved at the expense of an 8% frame loss.

# 6 Summary

This paper described the design of a software-only video effects processing system. The general architecture of the system is designed to work with commodity general-purpose processors connected by a high-bandwidth, low-latency network. The problem is to exploit functional, temporal, and spatial parallelism to construct real-time solutions for video effects processing.

The software architecture is composed of the FX Compiler, FX Mapper, and FX Processor. The FX Compiler constructs an intermediate dataflow representation of the desired video effect suitable for functional decomposition. The FX Mapper evaluates possible implementations of the dataflow graph and selects the one with the lowest expected cost. A set of FX Processors executes the video effect subprograms constructed by the FX Mapper.

Each type of parallelism requires the solution of different problems. The main problem with temporal parallelism is distributing video frames to processors and reassembling the processed video. Mechanisms were presented to solve these problems and their performance was measured.

Frame distribution (i.e., the selector function) is difficult to do in a decentralized manner. A centralized selector function was built and its performance was shown to be adequate, thereby precluding the need for a more complex decentralized solution. An algorithm for frame interleaving was presented that provided the ability to tradeoff latency and frame drops.

Measurements of the interleaver function show that the tradeoff can be successfully exploited.

The interleaver function uses information about arriving frames to dynamically adjust its behavior. We believe that this technique will be widely applicable in other mechanisms and can be used to generate control information to dynamically adjust and reconfigure other system components as well. For example, if frames that arrive at the interleaver from a particular FX Processor are consistently more delayed than frames that arrive from other FX Processors, the interleaver can communicate this information to the selector function to adjust how many frames are assigned to that processor.

## Acknowledgments

This work was supported by National Science Foundation grant #CDA-9512332 and equipment donated by Intel Corporation.

## References

[1] V. M. Bove, Jr. and J. A. Watlington. Cheops: A reconfigurable data-flow system for video processing. *IEEE Transactions on Circuits and Systems for Video Processing*, 5(2):140–149, April 1995.

[2] D. Chin, J. Passe, F. Bernard, H. Taylor, and S. Knight. The Princeton Engine: A real-time video system simulator. *IEEE Transactions on Consumer Electronics*, 32(2):285–297, 1988.

[3] David E. Culler et al. Parallel computing on the Berkeley NOW. *9th Joint Symposium on Parallel Processing*, 1997.

[4] D. A. Epstein et al. The IBM POWER Visualization System: A digital post-production suite in a box. *136th SMPTE Technical Conference*, pages 136–198, 1994.

[5] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, December 1997.

[6] T. Ikedo. A scalable high-performance graphics processor: GVIP. *Visual Computer*, 11(3):121–33, 1995.

[7] R. M. Lougheed and D. L. McCubbrey. The cytocomputer: a practical pipelined image processor. *Conference Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 271–278, 1980.

[8] Steven McCanne et al. Toward a common infrastructure for multimedia-networking middleware. *Proceedings of the 7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 1997.

[9] G. Millerson. *The Technique of Television Production.* Focal Press, Oxford, England, 1990.

[10] Anup Rao and Rob Lanphier. *RTSP: Real Time Streaming Protocol*, February 1998. Internet Proposed Standard, work in progress.

[11] Shigeru Sasaki, Tatsuya Satoh, and Masumi Yoshida. IDATEN: Reconfigurable video-rate image processing system. *FUJITSU Sci. Tech. Journal*, 23(4):391–400, December 1987.

[12] Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson. *RFC 1889, RTP: A Transport Protocol for Real-Time Applications*, January 1996.

[13] B. C. Smith. *Dali: High-Performance Video Processing Primitives.* Cornell University. Unpublished work in progress.

[14] B. C. Smith. *Implementation techniques for continuous media systems and applications.* PhD thesis, University of California, Berkeley : Computer Science Division, 1994.

[15] J. Swartz and B. C. Smith. RIVL: a Resolution Independent Video Language. *Proceedings of the Tcl/Tk Workshop*, pages 235–242, 1995.

[16] J.A. Watlington and V.M. Bove, Jr. A system for parallel media processing. *Parallel Computing*, 23(12):1793–1809, December 1997.

[17] T. Wong, K. Mayer-Patel, D. Simpson, and L. Rowe. Software-only video production switcher for the Internet MBone. *Proceedings of SPIE Multimedia Computing and Networking*, 1998.