Lag in Multiprocessor Virtual Reality

Matthias M. Wloka*
Computer Science Department
Brown University
Providence, RI 02912
mmw@cs.brown.edu

Abstract

Lag in virtual reality (VR), i.e., the delay between performing an action and seeing the result of that action, is critical when trying to achieve immersion. While multiple, networked processors have been used to increase through-put, we concentrate on using multiple processors to reduce lag. To that end, we present a complete list of all possible lag sources in VR applications, review available lag reduction techniques, and investigate how these reduction techniques interrelate. We also introduce a new process-synchronization scheme that reduces lag. We evaluate the effectiveness of this synchronization scheme both by software simulation, as well as by actual lag measurements in our sample application.

1 Introduction

Virtual reality (VR) applications strive to immerse users in artificial environments. To do this, common knowledge dictates a minimum frame rate of 10 frames per second. However, frame rate is only one of the important parameters that determine immersion: lag is equally important (Liu, Tharp, French, Lai, & Stark, 1993). Thus, instead of concentrating on frame rates (or through-put rates in general), system developers should aim more, we believe, to reduce lag (Bishop & Fuchs et al, 1992; Brooks, 1988).

1.1 What Is Lag?

Lag is the time between when a user performs an action and when the application displays the result of that action. For example, when a user moves a 3D input

*This work was supported in part by an IBM graduate fellowship for the author, the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, ONR Contract N00014-91-J-4052, ARPA Order 8225, and also Sun Microsystems, NASA, NCR, Hewlett Packard, and Digital Equipment Corporation.

device, various computation stages process and transform the 3D input data to make it visible on-screen, so that the user's movements are displayed with a finite delay. The process is illustrated in Figure 1.

Lag is important because human beings are extremely sensitive to it. For instance, depending on the task and surrounding environment, lag of as little as 100ms (less than a tenth of a second) degrades human performance (Held & Durlach, 1991; Liu, Tharp, French, Lai, & Stark, 1993). Even worse, if lag exceeds 300ms, humans start to dissociate their movements from the displayed effects, thus destroying any immersive effect (Held & Durlach, 1991).

This sensitivity to lag requires that we minimize lag for general applications to increase interactivity. VR applications in particular demand lag times of less than 300ms to uphold immersion. (We base this conclusion on the above cited results.) The most critical are augmented reality applications: for virtual objects in a see-through, head-mounted display to seem realistic, perceived lag must be less than 30ms (Held & Durlach, 1991).

Throughput measurements cannot substitute for lag measurements in assessing the interactivity of an application, since lag and throughput measure different quantities: lag measures how long a computation process delays data, and throughput measures how frequently a computation process delivers a result. However, the two quantities are related. Reducing lag in a computation process (for example, by using a faster algorithm) proportionally increases the throughput of that process (but only if one is able to sample new input data quickly enough). Yet increasing throughput does not necessarily decrease lag. For example, using multiple processors in a pipelined configuration increases throughput yet maintains the same lag.

In this paper, we focus on the general problem of lag for non-trivial interactions, i.e., responses to user input require substantial processing. Thus, tracking of head-mounted displays is a subproblem that is only marginally addressed, since it does not require substantial processing of the input.

We concentrate on lag, not throughput. First, we list and characterize all the sources of lag in VR applications. Second, we review techniques to reduce lag, in particular multiprocessing. We complement the review by pointing out the assumptions underlying each technique and how they interrelate. Third, we construct theoretical models of average lag and lag distributions that are verified by actual measurements; these models are crucial in quick evaluation of new lag reduction techniques for multiprocessing. Fourth and last, we introduce a new synchronization scheme that reduces total lag while maximizing throughput and is superior to previous schemes.

1.2 Single-CPU vs. Network-Parallel vs. MP-Workstation

Our discussion focuses on parallel VR applications. However, since all recent state-of-the-art VR systems take advantage of multiple processors, this is no restriction. Furthermore, since multiprocessor (MP) workstations are becoming increasingly available, we wish to take maximum advantage of their added computational power. This is not trivial: for example, we have found that using an MP workstation indiscriminately actually increases average end-to-end lag as compared to a single-CPU workstation (see Section 6)!

We distinguish a *multiprocessor (MP)* architecture from the *single-CPU* architecture and the *network-parallel* architecture as follows. A *single-CPU* architecture uses exactly one CPU (while additional, specialized compute power is usually available for rendering via a graphics card). This architecture contrasts with a *network-parallel* architecture, in which several CPUs are interconnected over a network and communicate via remote procedure calls or message passing. Typically, the network delays communication between CPUs nondeterministically by at least several milliseconds. Finally, a *multiprocessor* architecture incorporates a small number of CPUs into a single workstation; in this architecture the processes on the CPUs typically communicate via shared memory without (measurable) delay.

The results we present here are not limited to the MP architecture. The discussions of the sources of lag, lag reduction techniques, and theoretical lag models are general enough to apply equally well to network-parallel applications and even single-CPU VR applications. Only the various synchronization schemes, by their very nature, do not apply to single-CPU applications. Where appropriate, we point out differences between MP and network-parallel architectures and how they influence corresponding conclusions.

1.3 Overview

We review previous work in Section 2, and we analyze and characterize all possible sources of lag in a typical VR application in Section 3. In Section 4, we review lag reduction techniques, namely prediction, time-critical computing, and parallelism, and investigate what interrelating assumptions these techniques make. Section 5 describes our implementation and theoretical models to measure and compare lag. In Section 6, we introduce and discuss a new synchronization scheme that minimizes lag. Finally, we draw conclusions from our experiments and suggest possible future work in Section 7.

2 Previous Work

So far, most research in parallel VR systems aims at increasing throughput, not decreasing lag. A report by Mark Mine (Mine, 1993) is an exception. However, while Mine realizes the importance of lag in head-mounted display systems, his description of its sources is insufficient. He omits synchronization delay as a source of lag and fails to generalize his characterization of the other sources. Most of his report concentrates on measuring user input device delays. And while Adelstein et al (Adelstein, Johnston, & Ellis, 1992) also quantify delays of user input devices, their results have only limited impact on total end-to-end lag in VR applications, because they describe lag of user input devices in isolation.

Liang et al (Liang, Shaw, & Green, 1991) claim to measure user input device delay, even though they actually measure end-to-end lag (albeit for an virtual environment with very simple graphics). Nonetheless, due to the focus of their work, the same criticism applies: they attempt to isolate lag of user input devices and disregard other sources of lag.

Other researchers describe methods to reduce lag, such as prediction, time-critical computing, and use of parallelism. In particular, Liang et al (Liang, Shaw, & Green, 1991), Friedman et al (Friedmann, Starner, & Pentland, 1992), and Deering (Deering, 1992) propose prediction of user input data as a way to reduce and even eliminate perceived lag. Deering implements linear extrapolation, while the others apply a Kalman filter which provides the best least squares predictor in the presence of Gaussian noise. However, the authors fail to describe all the underlying constraints that restrict the use of prediction in VR applications (see Section 4.1).

Time-critical computing (Funkhouser & Sequin, 1993; Gossweiler, 1993; Holloway, 1991; Wloka, 1993) allows "computing to a budget." In conventional computing paradigms, we wait for the computer to finish its tasks, however long it takes. Time-critical computing ensures that we have at least approximate results at certain time-deadlines. Thus, time-critical computing bounds the maximum lag. We believe we are the first to explore the benefits of using time-critical computing in connection with user input prediction to control lag.

The increased compute power of parallel processing is attractive to VR system designers, and thus several VR systems are network-parallel or MP-parallel (Appino, Lewis, Koved, Ling, Rabenhorst, & Codella, 1992; Codella, Jalili, Koved, & Lewis, 1993; Gobbetti, Balaguer, & Thalmann, 1993; Lewis, Koved, & Ling, 1991; Shaw, Liang, Green, & Sun, 1992; Wang, Koved, & Dukach, 1990). All these systems use asynchronous or other *ad hoc* process communication (except for left- and right-eye view rendering). Yet asynchronous communication maximizes throughput but is suboptimal for reducing lag. While Appino et al (Appino, Lewis, Koved, Ling, Rabenhorst, & Codella, 1992) mention the possibility of "just-in-time" synchronization schemes, they do not actually detail or implement any such schemes. We are not aware of any work that analyzes and optimizes synchronization schemes to reduce lag.

3 Lag Sources in VR Systems

3.1 User Input Device Lag

The user input device in a VR application reports 3D position and orientation data. It is external to the host workstation and typically communicates data via the serial port. Examples of such devices include the Logitech "Red Baron" ultrasound sensor, the Polhemus Isotraks, and the Ascension Bird. Total user input device lag includes signal generation and communication time.

Depending on the type of device and mode of operation (i.e., noise filtering on/off, different orientation reporting modes, etc.), lag ranges from 10ms to 120ms; throughput is between 30 and 50 samples per second (Adelstein, Johnston, & Ellis, 1992; Mine, 1993),

As a specific example, the Logitech "Red Baron" sensor reporting position and Euler angles in a small work-volume in streaming mode has a lag of 47.5ms. Throughput for this device is 50 samples per second, i.e., one sample every 20ms. (We made these measurements using our setup (see Section 5); Logitech independently confirmed them.)

3.2 Application-Dependent Processing Lag

Once the user input device data arrives at the host workstation, the application processes it. Processing can be as simple as transforming the data from the device format to the rendering format, i.e., the application echoes the user input device position to the virtual environment. Other more complicated application processes are common, for example, interactive streamline computations for virtual wind-tunnels (Bryson & Levitt, 1991).

Processing lag is highly application-dependent and thus highly variable. The simple echoing scheme above is the lower bound; today's workstations perform these data transformations in one millisecond or less. The upper bound is harder to characterize. However, keeping in mind that the resulting VR system is supposedly immersive, we assume that the lag introduced by application processing does not exceed 500ms, since it is unlikely that applications with input processing requirements beyond 500ms can be made immersive. Therefore, application-dependent processing lag ranges from 1 to 500ms.[1]

Throughput of the application depends on the number of processors available. In the single-CPU case, the same processor computes the application and also feeds the renderer (see Section 3.3). Therefore, throughput is

$$\frac{1000}{(application\_lag + render\_lag)} \text{ times per second,}$$

with all lag times measured in milliseconds.

With at least two processors available, we assign one to feed the renderer. Application throughput is thus at least $(1000/application\_lag)$ times per second, which translates to at least twice per second.

If more than two processors are available, the application task should first be parallelized so as to reduce lag. If thereafter application throughput is still

[1] Of course, application delays of 500ms are only permissible if head-tracking proceeds asynchronously and independently with considerably less lag and higher frame rates; otherwise immersion is not achievable.

worse than rendering throughput (see Section 3.3) and processors are idle, then we recommend running several instances of the application on different user input data until application throughput is equal to or better than rendering throughput. Since rendering throughput is at most 72 times per second (see Section 3.3), application throughput is thus 2 to 72 times per second.

3.3 Rendering Lag

Rendering lag is the time from sending data to the rendering hardware until the same data is displayed on the monitor. We assume double-buffering rendering hardware that does not use the CPU for rendering computations. Since double-buffering synchronizes the rendering hardware with the display refresh, the finite display refresh rate (typically 60-72Hz) causes a minimum rendering lag of 14ms. The maximum rendering lag derives from the minimum requirement of 10 frames per second: 100ms. Rendering lag is highly scene- and viewpoint-dependent, and thus is likely to vary during the run-time of an application.

The maximum rendering lag is longer if the rendering hardware is heavily pipelined. For instance, the 1,000,000 polygons/sec Pixel-Planes 5 architecture (Fuchs, Poulton, Eyles, Greer, Goldfeather, Ellsworth, Molnar, Turk, Tebbs, & Israel, 1989) renders a single polygon in 54ms (Mine, 1993). (However, Pixel-Planes 5 can be reconfigured to allow for shorter lag at the expense of throughput.) Our measurements indicate that conventional graphics cards (in particular, the Sun ZX graphics boards) do not exhibit this anomaly.

The scan-out of the display, since it occurs with a frequency of 60 to 72Hz,[2] causes additional lag. Depending on where the rendered data appears on the display and whether the display refreshes from top to bottom or vice versa, the data image may remain invisible for a further 0 to 17ms.

As in the application case, rendering throughput depends on the number of processors available. If only a single processor is available, rendering throughput equals application throughput, i.e., $1000/(application\_lag + render\_lag)$ times per second, since the single CPU computes the application and also feeds the renderer.

If at least two processors are available, assigning one of them exclusively to feed the renderer yields a rendering throughput of $1000/render\_lag$ times per second. Since we also assume that only a single graphics board renders into the frame-buffer, the presence of additional processors cannot further influence rendering throughput.

3.4 Synchronization Lag

Parallel VR applications process user input in several stages: the user input device processing stage, the application-dependent processing stage, and the

[2]These values apply to CRT displays. LCD-type displays have different characteristics.

rendering stage.[3] Since these stages are independent, it is possible (and in fact likely) that, for example, the user input device deposits a new sample on the serial port shortly after the application reads the serial port. Thus, the application is busy processing the previous input before it reads the serial port again and starts to process the current input, so that user input data is delayed because it is waiting to be processed by a currently busy stage.

We define synchronization lag as the total time data is waiting in-between stages without being processed.[4] Synchronization lag is thus inversely proportional to the throughput rates of the various stages. It also varies during the run-time of the application.

In the best case, synchronization lag is zero: each stage writes its output just before the next stage reads the data. The worst case is equally likely: each stage writes its output just after the next stage reads the data. Synchronization lag thus varies from 0 to a maximum of the sum of the inverse throughput rates of each stage. On average, synchronization lag is half that maximum, so that average synchronization lag varies depending on the throughput rates of the various stages, i.e., it varies from

$$\left(\frac{1000}{\text{max\_throughput\_UID}}+\frac{1000}{\text{max\_throughput\_appl}}+\frac{1000}{\text{max\_throughput\_render}}\right)/2 =$$

$$(20 + 15 + 15)/2 = 25\text{ms}$$

to

$$\left(\frac{1000}{\text{min\_throughput\_UID}}+\frac{1000}{\text{min\_throughput\_appl}}+\frac{1000}{\text{min\_throughput\_render}}\right)/2 =$$

$$(33 + 500 + 100)/2 = 316.5\text{ms}.$$

While synchronization lag is easy to overlook, it contributes up to 50% of the total lag in a VR system.

3.5 Frame-Rate-Induced Lag

Slow frame rate induces a sample-and-hold artifact that has characteristics similar to lag. The moment we display a new frame, and thus new data, the data on-screen is as up-to-date as possible. However, as time goes on and the display is not updated, the data displayed becomes progressively out of date. We call this phenomenon *frame-rate-induced lag*; it depends only on the frame rate and thus its maximum ranges from 15 to 100ms.

We distinguish frame-rate-induced lag from all other lag sources. We define *end-to-end lag* as the delay between when the user moves the user input

[3] Even in the single-CPU case, the user input device is separate and independent from the CPU computing the application and feeding the renderer. Thus, VR applications process user input in two stages when running on a single-CPU architecture.

[4] In the network-parallel case, synchronization lag also includes the network delays.

device and the first display of that movement. End-to-end lag thus includes user input device lag, application-dependent processing lag, rendering lag, and synchronization lag, but specifically excludes frame-rate-induced lag.

Frame-rate-induced lag is one of the reasons that slow frame rates are unacceptable for VR applications. It is therefore important to quantify frame-rate-induced lag in correlation to end-to-end lag. Yet simply adding the maximum frame-rate-induced lag to end-to-end lag, i.e., adding the time a frame is on-screen to the end-to-end lag, is insufficient: such a model would imply that the user first sees the data only when it is about to be replaced by newer data. Instead, we model the human visual system as another processing stage that interfaces the VR application display to the brain. This processing stage reads the display immediately after a new frame is displayed. Let us call the time this new frame is displayed t_{new}. Thus, at time t_{new} we register only the end-to-end lag l.

Only after a finite time-interval i_{perc} can the human eye receive new data, i.e., the human visual system has limited bandwidth. Thus, at time $t_{new} + i_{perc}$ we register a total lag equal to the end-to-end lag plus the age of the on-screen data: $l + i_{perc}$. We repeat this process, recording lag times of $l + k \cdot i_{perc}$ for $k = 0, 1, \ldots$, until a new frame is displayed, i.e., until

$$k \cdot i_{perc} > \text{ time that the frame (data) is on-screen.}$$

Since the human visual system perceives flicker on video monitors only up to a rate of about 70Hz (Blaire-Benson, 1986), we conclude that the above time interval i_{perc} is roughly equal to $1000/70 = 14.3$ms. To be safe we assume i_{perc} to be equal to 5ms.

This model successfully combines end-to-end lag and frame-rate-induced lag, while also being consistent with the test results of others (Liu, Tharp, French, Lai, & Stark, 1993). We rely on it when comparing different lag-reduction techniques in Section 6, since some of these techniques influence both end-to-end lag and frame rate and thus frame-rate-induced lag.

Figure 1 summarizes the findings of this section.

4 Techniques to Reduce Lag

4.1 Prediction

Prediction methods extrapolate past user input data to future time points, thus reducing perceived lag (Deering, 1992; Friedmann, Starner, & Pentland, 1992; Liang, Shaw, & Green, 1991). However, this extrapolation process introduces spatial inaccuracies that increase under the following three conditions (Friedmann, Starner, & Pentland, 1992; Liang, Shaw, & Green, 1991): (1) the user input device throughput is too low; (2) we predict too far into the future; (3) the user input device acceleration is too high.

Yet prediction is the only available method that can drastically reduce total perceived lag and in particular application-dependent processing lag (since we are discussing the general problem of transforming the user input non-trivially in the application-dependent processing stage). To minimize lag, the user input device stage projects the user input data to the time this data reaches the display. Thus, the user input device stage requires knowledge about the lag experienced by the predicted data in future stages. Prediction thus demands constant (or close to constant) application-dependent processing lag and rendering lag, as well as synchronization lag with as narrow a distribution as possible. In general, even prediction cannot eliminate perceived lag, because of variations in total end-to-end lag. We illustrate these requirements in Figure 2.

4.2 Time-Critical Computing

It is not advisable to use time-critical computing (Funkhouser & Sequin, 1993; Gossweiler, 1993; Holloway, 1991; Wloka, 1993) directly to reduce lag. Since time-critical computing trades computation time for computation accuracy, saving maximum time by computing with the least accuracy would produce gross visual errors while still not fully eliminating lag. The benefit gained is questionable.

We propose instead to use time-critical computing to assure constant or nearly constant application-dependent processing lag and rendering lag. This brings us one step closer to being able to use prediction, as shown in Figure 2.

4.3 Multiple Processors

Multiple processors reduce lag in a VR application in several ways. If we parallelize the application process, we can reduce application-dependent processing time directly. Pipelining the application or running several instances of it increases the throughput of the application, and thus decreases the expected average synchronization lag of data waiting to be processed by the application. However, the most popular use of multiple processors is to assign at least one to each computation stage.

Using at least one CPU for each computation stage in a VR application, even in asynchronous communications mode (Appino, Lewis, Koved, Ling, Rabenhorst, & Codella, 1992; Codella, Jalili, Koved, & Lewis, 1993; Lewis, Koved, & Ling, 1991; Shaw, Liang, Green, & Sun, 1992; Wang, Koved, & Dukach, 1990), has four main advantages. First, the user input device is independent from all other stages and thus runs with maximum throughput, allowing use of prediction (see Figure 2). Second, rendering also proceeds at maximum throughput, reducing frame-rate-induced lag (see Section 3). Third, the distribution of synchronization lag is also narrower and thus better than in the single-CPU case. Fourth and finally, by allowing the user-input processor to communicate user input device data directly to the rendering stage, we can “short-circuit” the application and display a low-lag cursor echoing the user input device position directly in addition to the high-lag application-computed feedback.

In Section 6 we introduce better synchronization schemes that, while maintaining all the above advantages of parallel processing, also reduce average synchronization lag. Surprisingly, asynchronous communication actually increases average synchronization lag over the single-CPU architecture.

5 Measuring Lag

5.1 Implementation and Measuring Lag in Practice

To evaluate lag-reduction techniques, we implemented a simple VR application: we pass a wand that emits several streamlines through a data-set representing the airflow around the space shuttle. The interactively computed streamlines let us visualize the flow.

The application reads the position and orientation of a user input device, processes the data by computing streamlines of the flow field for that position and orientation, and renders these streamlines. We also render a Gouraud-shaded representation of the space shuttle consisting of approximately 9000 triangles. A single Logitech "Red Baron" ultrasound device mediates user input to a four-processor Sparc 10 workstation with a ZX graphics board. The Logitech device runs in streaming mode, reporting Euler angles in the small work-volume.

We use only three of the four processors available: the first processor continuously scans the serial port for new user input and extrapolates it to future times, the second processor is responsible for computing the streamlines, and the third feeds the rendering hardware. All processors run asynchronously (and later synchronously, according to the synchronization scheme) and each stage simply overwrites previous output, i.e., no buffering occurs. We leave the fourth processor idle to handle spontaneously occurring operating system (SunSoft Solaris) or other systems-related tasks. Thus, we ensure that the operating system never swaps out any of our application tasks.[5]

We measure end-to-end lag of our implementation in a setup practically identical to that described by Mine (Mine, 1993): we use an oscilloscope to determine the time difference between when the Logitech tracker crosses a fixed boundary and when the computer graphically acknowledges the crossing by displaying a white triangle. Our measurements (see Figure 3) eliminate display scan-out lag (see Section 3.3) since we display the triangle so that it is scanned out first.

[5]The serial port causes additional systems-related lag. Standard Solaris only processes the serial port every 30ms, unnecessarily introducing 0 to 30ms additional lag. We modified the serial port routines to eliminate this lag.

5.2 Evaluating and Comparing Lag in Theory

Implementing a new synchronization scheme and measuring the resulting lag is time-consuming and cumbersome. We have therefore developed theoretical lag models that let us calculate the expected lag (and thus usefulness) of a given scheme. The models below estimate average lag as well as lag distribution.

5.2.1 Orbit Model

In the *orbit model*, we simulate in software the behavior of our application. *Events* represent user input motion: every time the user moves the user input device, a new event is generated. We assume the user moves every millisecond,[6] and thus each millisecond we generate a new event. Our software simulation tracks the creation time of each event.

We then simulate an event's passage through the various processing stages. The user input device stage, the application-dependent processing stage, and the rendering stage each delay an event by a constant amount of time. An event is further delayed if, upon arrival at one of these stages, the stage is busy processing previous events, i.e., the event experiences synchronization lag. Each event therefore influences the lag of successive events, since once a stage starts to process an event, all successive events must wait until it becomes idle again. When an event exits the last stage, i.e., is displayed, we compare its exit time to its creation time to determine its end-to-end lag.

Since we assume that processing time for the user input device stage, the application-dependent processing stage, and the rendering stage is constant, the synchronization lag before each stage fully characterizes the end-to-end lag of an event. We thus notate an event as a tuple of numbers corresponding to the synchronization lag experienced before each stage. In the single-CPU case, an event is thus a pair, since there are only two stages; in general, there are three stages and an event is a triple.

While this tuple-notation characterizes an event, it also describes the period during which each stage is busy after the arrival of that event. Thus, it fully determines the synchronization lag of the next event. Each event therefore determines a *string* of successive event lag times.

Since each stage has finite and constant processing time, events can experience only a finite number of different synchronization lags before each stage. Thus, the space of non-identical (in the tuple notation sense) events is finite. Accordingly, each event-originated string must form a cycle after a finite number of events — we call the generated circular structure of an event-string an *orbit* (we exclude the appendages that lead into an orbit). Figure 4 shows examples.

[6] This assumption is not strictly correct: human motion is continuous. However, compared to the much slower Logitech tracker sampling rate, a rate of 1kHz approximates the continuous behavior well enough.

The orbit model records the end-to-end lag of the statistically relevant events. We find these relevant events as follows. The model loops through all possible synchronization lag combinations to generate all possible starting events. Each starting event computes its limit orbit. All the events that are part of an orbit are relevant. (All events that are part of an appendage are not relevant, however, since they rarely occur in practice.)

Since all the events in an orbit, as well as the events in the appendages leading into that orbit, generate that same orbit, we record the end-to-end lag of the events in that orbit as many times as the orbit was generated. The more often an orbit is generated the more statistically relevant it is. Thus, the end-to-end lag of the events in a long orbit with many appendages is recorded many more times than the end-to-end lag of the events in a short orbit with no appendages.

5.2.2 Combination Model

The orbit model estimates end-to-end lag as defined in Section 3. It does not take into account frame-rate-induced lag. The *combination model* simulates and measures frame-rate-induced lag via the model described in Section 3.5.

The combination model is an extension of the orbit model. Instead of identifying an event by its synchronization lag, the combination model identifies events by their synchronization lag and the amount of time each is on-screen. As before we generate all possible strings and count as relevant only the events that are part of orbits. We then convert the on-screen time of each relevant event into frame-rate-induced lag as described in Section 3.5. That is, each relevant event with end-to-end lag l and on-screen time t spawns a set of lag times

$$\{l + k \cdot i_{perc} \mid k = 0, 1, \ldots, \frac{t}{i_{perc}}\}.$$

(The constant i_{perc} corresponds to the finite throughput of the human visual system as explained in Section 3.5, and is set equal to 5ms.) We add all the lag times in these sets to the lag distribution.

6 Minimizing Synchronization Lag

Synchronization lag is one of the largest sources of lag in a typical VR application. We introduce a new synchronization scheme that reduces average synchronization lag, narrows lag distribution, and maintains high user input device and rendering throughput. Thus, this new synchronization scheme is attractive for MP VR applications in general and predictive MP VR applications in particular.

We compare this new synchronization scheme to the single-CPU case and the asynchronous synchronization scheme, and evaluate these comparisons using the orbit and the combination model (see Section 5.2). To validate the evaluations we also show the actual, measured end-to-end lag of our implementation (see Section 5.1) for each scheme.

6.1 Single-CPU Synchronization

We implement the single-CPU case on our four-processor workstation by using only one computation thread. This single thread reads the serial port, computes the application, and sends data to the rendering hardware. The average end-to-end lag is better than the expected

$$48 + (43 + 61) + \frac{20 + 104}{2} = 214 \text{ ms},$$

because the rendering hardware computes partially in parallel with the CPU (refer to Figure 3 for the sources of the various delays). Thus, we adjust application and rendering throughput from the expected $(1000/(43 + 61)) = 9.6$ frames per second to the actually measured 11.0 frames per second. The results are shown in Figure 5, Figure 9, and Table 1.

6.2 Asynchronous Synchronization

Asynchronous synchronization assigns one processor to read the serial port continuously, one to compute the application process, and one to feed the renderer. Each processor runs independent of all others, and thus at maximum throughput. The resulting lag times are shown in Figure 6, Figure 9, and Table 1.

6.3 Just-in-Time Synchronization

Just-in-time synchronization is a new synchronization scheme. Like asynchronous synchronization, it assigns one processor to read the serial port continuously, one to compute the application process, and one to feed the renderer. The processor that reads the serial port runs as fast as possible, thus producing maximum throughput for the user input device. Similarly, the processor that feeds the renderer also runs as fast as possible to maintain maximum rendering throughput.

Unlike asynchronous synchronization, however, the application process does not run as often as possible. Instead, we start the application process so that it finishes computing just before the rendering process starts to compute a new frame. Thus, the rendering process always renders application data that is as up-to-date as possible.

This synchronization scheme relies on two assumptions: first, that we know the computation time the application process requires, and second, that we know when the rendering process starts to read data. That is we assume knowledge of the rendering time of the currently rendering frame.

Time-critical computing ensures that both assumptions are correct. In particular, in our sample implementation we compute the streamlines time-critically. This means that the application process computes for 43ms, and upon reaching this limit returns and communicates the result to the renderer. On the other hand, the renderer in our application has a fairly constant load, thus obviating advanced time-critical rendering techniques: it is sufficient to monitor rendering times for each frame and use a weighted average of these to predict the rendering time of the current frame.

To make just-in-time synchronization robust against unpredicted delays, we adjust it slightly. Our simulations show (and our measurements confirm) that it is advantageous for the application stage to wait for fresh data from the user input stage. Instead of accepting the user input data indiscriminately, the application process examines the age of the currently available user input device data, i.e., when it was written. If it is older than 10ms, the application process waits for fresh data.[7] Since the user input device updates every 20ms, the application process waits less than 10ms. Accordingly, the rendering process checks the age of the application data. If the data is older than 10ms, the rendering process waits for new data to arrive. Because the application delays data by no more than 10ms, the maximum delay for the rendering stage is 10ms.

While this adjustment decreases rendering throughput slightly, the advantage of always processing fresh data balances the lost time (see Figure 8). More important, the synchronization scheme becomes robust against prediction errors of the rendering times.[8]

Figure 7, Figure 9, and Table 1 show the resulting lag distributions and performance of the just-in-time synchronization scheme.

6.4 Comparison of Synchronization Schemes

As Figure 9 and Table 1 clearly show, just-in-time synchronization outperforms single-CPU and asynchronous synchronization. In particular, just-in-time synchronization reduces synchronization lag by about 33% as compared to asynchronous synchronization, and reduces average total lag in an MP VR application by about 10%. Just-in-time synchronization also narrows the spread of lag times for individual samples, while maintaining maximum throughput for the user input device. Finally, just as in the asynchronous case, just-in-time synchronization lets us "short-circuit" (see Section 4.3) the application.

However, just-in-time synchronization has disadvantages. It lowers the frame rate slightly. However, Liu et al (Liu, Tharp, French, Lai, & Stark, 1993) argue that such a performance degradation is insignificant. Just-in-time synchronization is also harder to implement than asynchronous synchronization, and it

[7] If the application relies on more than one input device, the same general mechanism is applicable, using as the age a weighted average of the ages of all input devices.

[8] This scheme is also robust against unpredictable network delays in the network-parallel case, so that just-in-time synchronization should be applicable there as well.

requires that computation times of both the application process and the rendering process be predictable. On the other hand, predictability or at least the existence of an upper bound for application and rendering times is generally desirable for VR applications, since otherwise user input prediction cannot be used to the fullest advantage, and worse, immersion would be lost due to excessive lag times for individual frames.

7 Conclusions and Future Work

From the findings above, we recommend the following way to use multiple processors to reduce lag in VR applications. First, assign processors to user input devices. While previous work (Appino, Lewis, Koved, Ling, Rabenhorst, & Codella, 1992) recommends one processor per user input device, we modify this recommendation. Since serving a user input device has low overhead (we only need to read the serial port every 20ms and predict), one processor suffices to serve several user input devices. Achieving maximum throughput for the user input devices enables maximum performance of the prediction algorithm (see Section 4) and thus minimizes perceived lag.

We reserve one processor for each hardware rendering board: since rendering is typically the bottleneck in VR applications, we need to utilize the available rendering hardware maximally.

All remaining processors process the application stage, which should be parallelized to reduce lag as much as possible. Once lag cannot be reduced any further, processors should run separate instances of the application to increase its throughput to match the rendering stage's throughput.[9]

Rendering as well as application processing should be time-critical to bound the maximum possible lag. The just-in-time synchronization scheme should then be used to provide data communication between the various stages in order to minimize average end-to-end lag as well as lag variation. Since the setup described fulfills all requirements of prediction, predicting user input data to the minimum expected end-to-end lag is advisable to minimize perceived lag.

In our sample application running on an MP architecture, we also reserve one processor to run spontaneously occurring OS tasks. Leaving a processor idle is not required if the user threads are locked into individual processors (thus avoiding OS-internal rescheduling) or if the application runs on a network-parallel architecture.

The effects on lag of the operating system preempting and rescheduling user tasks on an overloaded MP-architecture are worth studying. In particular, how does average lag and lag distribution change? We found that once Solaris preempts a user thread, it only reschedules it on the order of 10ms later. Does the added processing power of using the idle processor balance the lag introduced

[9]Pipelining achieves the same effect. However, since proper load balancing is harder to achieve, we discourage use of pipelining for the application process.

by processes being swapped out? How does process rescheduling influence user input prediction and compute time predictions used in the time-critical algorithms?

Other future work includes extending our sample VR application to operate on dynamic data sets; this will require further research in the general areas of time-critical computing and time-critical rendering. Furthermore, the lag models introduced in Section 5 simulate actual lag only. Since prediction trades spatial errors for better temporal accuracy, thus reducing perceived lag, models that quantify this trade-off and thus simulate perceived lag are needed. User studies should prove helpful in this task.

Acknowledgements

We thank Nate Huang and Tom Meyer for implementing parts of the example application, Bob Zeleznik for helpful discussions, our Sun sponsor Barry Medoff for speedy resolution of hardware-specific questions, Michael Deering for providing the code that reads and predicts the user input device, and the anonymous reviewers for the many detailed comments.

References

Adelstein, B. D., Johnston, E. R., & Ellis, S. R. (1992). A testbed for characterizing dynamic response of virtual environment spatial sensors. *1992 UIST Proceedings*, 15–22.

Appino, P. A., Lewis, J. B., Koved, L., Ling, D. T., Rabenhorst, D. A., & Codella, C. F. (1992). An architecture for virtual worlds. *Presence*, 1(1), 1–17.

Bishop, G., & Fuchs, H., et al (1992). Research Directions in Virtual Environments: Report of an NSF Invitational Workshop, March 23-24, 1992, at UNC Chapel Hill. *Computer Graphics*, 26(3), 153–177.

Blaire-Benson, K. (1986). *Television Engineering Handbook*, chapter 1. McGraw Hill.

Brooks, Jr., F. P. (1988). Grasping reality through illusion – interactive graphics serving science. In *Human Factors in Computing Systems*, 1–11. Special Issue of the SIGCHI Bulletin.

Bryson, S., & Levitt, C. (1991). The virtual windtunnel: An environment for the exploration of three-dimensional unsteady flows. In *Visualization '91*, 17–24.

Codella, C. F., Jalili, R., Koved, L., & Lewis, J. B. (1993). A toolkit for developing multi-user, distributed virtual environments. In *IEEE Virtual Reality Annual International Symposium*, 401–407.

Deering, M. (1992). High resolution virtual reality. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2), 195–202.

Friedmann, M., Starner, T., & Pentland, A. (1992). Device synchronization using an optimal linear filter. *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, 25(2), 57–62.

Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., & Israel, L. (1989). Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics (SIGGRAPH '89 Proceedings)*, 23(3), 79–88.

Funkhouser, T. A. & Sequin, C. H. (1993). Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proceedings)*, 247–254.

Gobbetti, E., Balaguer, J.-F., & Thalmann, D. (1993). VB2: An architecture for interaction in synthetic worlds. In *1993 UIST Proceedings*, 167–178. ACM SIGGRAPH, Addison-Wesley.

Gossweiler, R. (1993). Time-critical rendering in an immersive virtual environment. Thesis Proposal, available from author on request.

Held, R., & Durlach, N. (1991). Telepresence, time delay and adaptation. In Stephen R. Ellis, editor, *Pictorial Communication in Virtual and Real Environments*, chapter 14. Taylor and Francis.

Holloway, R. L. (1991). Viper: A quasi-real-time virtual worlds application. Technical Report TR92-004, University of North Carolina, Chapel Hill.

Lewis, J. B., Koved, L., & Ling, D. T. (1991). Dialogue structures for virtual worlds. *Proceedings of CHI'91*, 131–136.

Liang, J., Shaw, C., & Green, M. (1991). On temporal-spatial realism in the virtual reality environment. *Proceedings of the 1991 User Interface Software Technology*, 19–25.

Liu, A., Tharp, G., French, L., Lai, S., & Stark, L. (1993). Some of what one needs to know about using head-mounted displays to improve teleoperator performance. *IEEE Transactions on Robotics and Automation*, 9(5), 638–648.

Mine, M. R.. (1993). Characterization of end-to-end delays in head-mounted display systems. Technical Report TR93-001, University of North Carolina at Chapel Hill.

Shaw, C., Liang, J., Green, M., & Sun, Y. (1992). The decoupled simulation model for virtual reality systems. *Proceedings of CHI'92*, 321–328.

Wang, C. P., Koved, L., & Dukach, S. (1990). Design for interactive performance in a virtual laboratory. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2), 39–40.

Wloka, M. M. (1993). Ph.D. thesis proposal: Time-critical graphics. Technical Report CS-93-50, Brown University, Department of Computer Science, Providence, RI.

		Asynchronous / Single-CPU	Just-in-Time / Single-CPU	Just-in-Time / Asynchronous
Measured Data	sync. lag	1.22	0.76	0.62
	avg lag	1.05	0.89	0.85
	spread	0.78	0.75	0.87
	.9-spread	0.90	0.64	0.71
Orbit Model	sync. lag	1.11	0.75	0.67
	avg lag	1.03	0.93	0.91
	spread	1.04	0.88	0.85
	.9-spread	0.88	0.58	0.66
Combination Model	avg lag	0.96	0.88	0.92
	spread	0.82	0.77	0.94
	.9-spread	0.75	0.71	0.95

Table 1: We use our actual measured data, the orbit model, and the combination model to compare the average synchronization lag (sync. lag), the overall average lag (avg lag), the spread (spread), and the .9-spread (.9-spread) of the different synchronization schemes. The just-in-time synchronization scheme reduces synchronization lag by 33% as compared to asynchronous synchronization. Total average lag in our VR application is reduced by about 10% as compared to asynchronous synchronization.

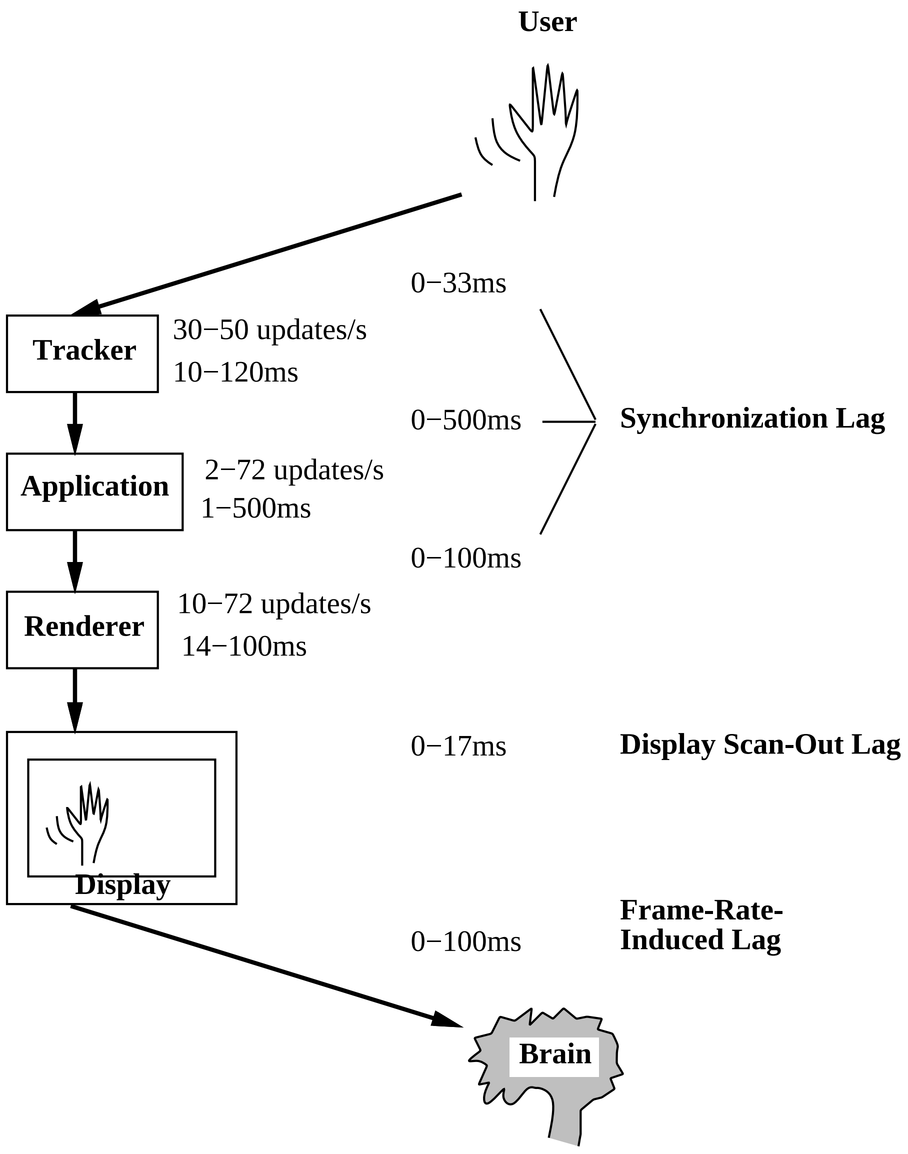

Figure 1: A typical VR application reacts to a user's actions with a finite delay caused by several characteristic components.

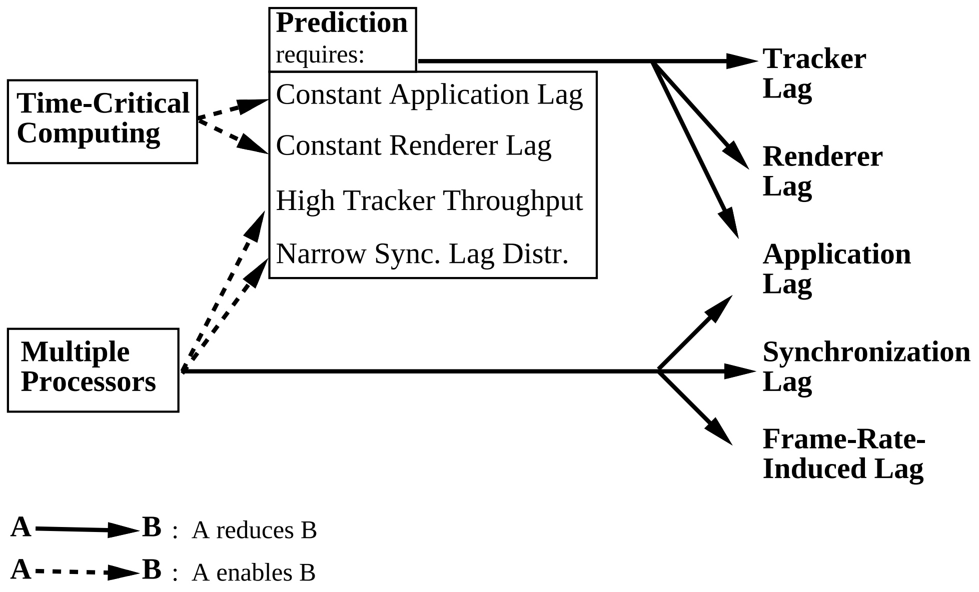

Figure 2: Interdependencies of the various lag-reduction techniques.

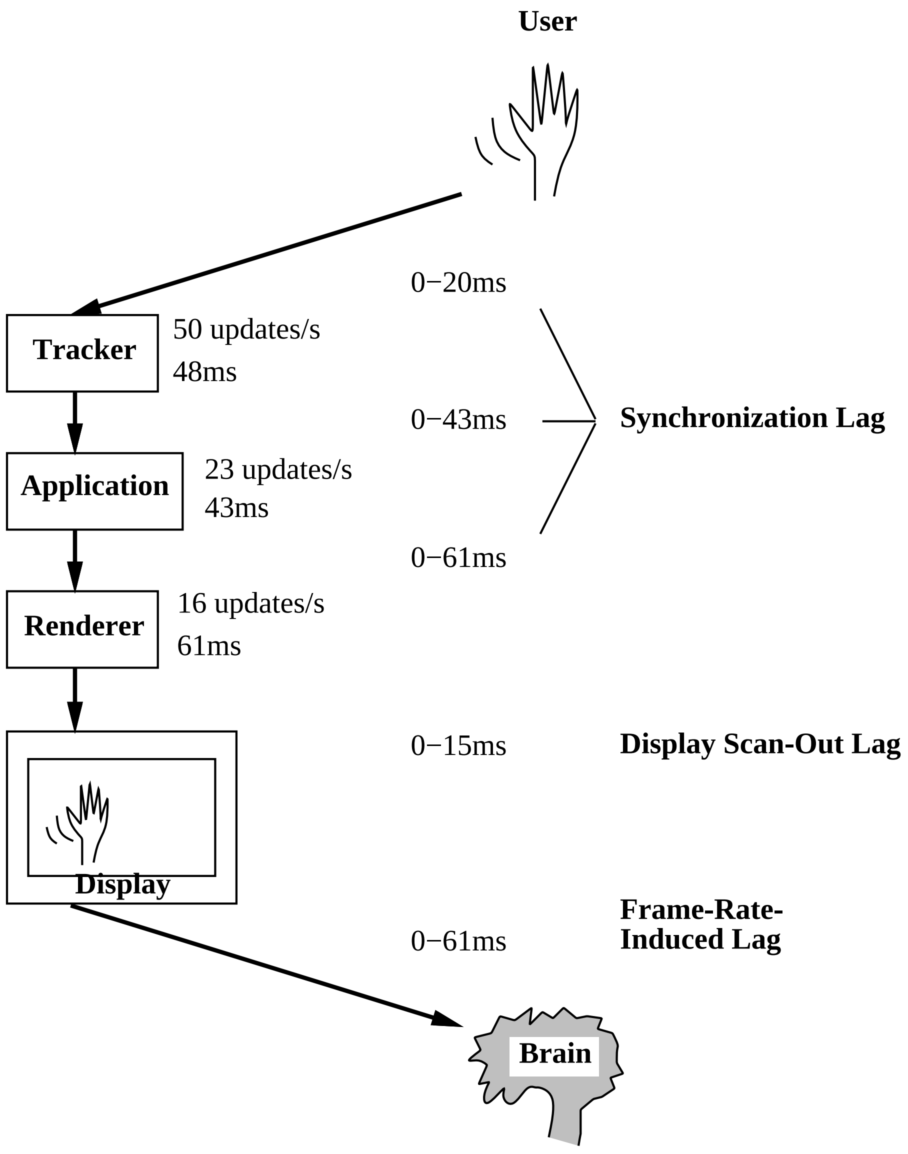

Figure 3: The particular lag times of our sample implementation.

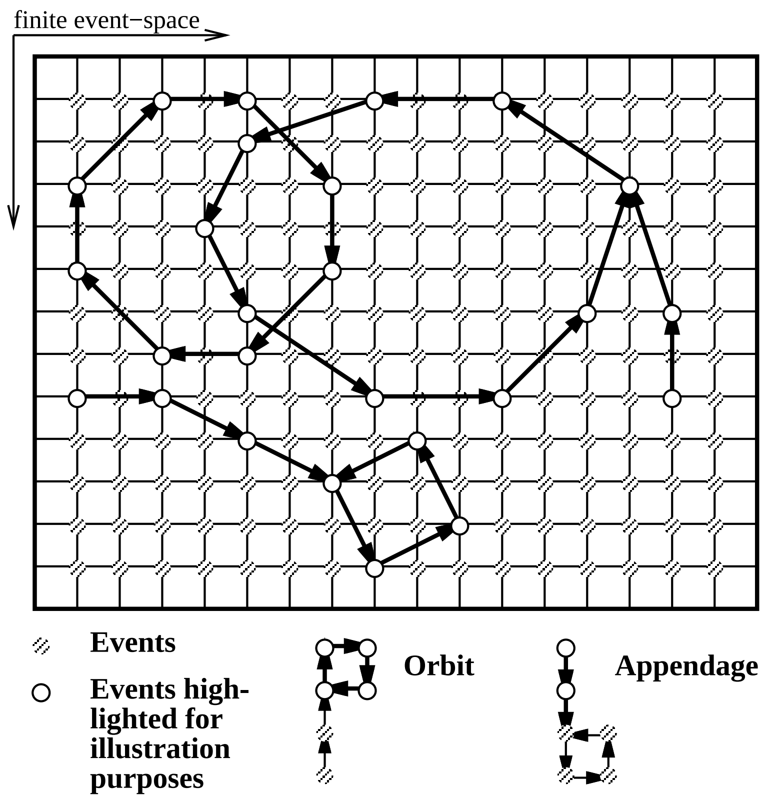

Figure 4: With constant processing times for each stage, each event determines the end-to-end lag of its successor. Therefore, each event generates a string of succeeding events, as shown here. Because the number of combinations of different lag-times for each event is finite, the strings ultimately circle, i.e., they form orbits, giving the orbit model its name. The examples shown here do not represent real processes; they are chosen arbitrarily for illustration purposes.

Figure 5: Lag distribution for the single-CPU case as estimated by the orbit model is compared with the actual, measured end-to-end lag. The end-to-end lag of 200 samples was measured. The spread is the maximum deviation from the average lag; the .9-spread indicates maximum deviation from the average lag, taking into account only the 90% of the total number of samples that are closest to the average.

Figure 6: Lag distribution for asynchronous synchronization as estimated by the orbit model is compared with the actual, measured end-to-end lag. The end-to-end lag of 200 samples was measured. The spread is the maximum deviation from the average lag; the .9-spread indicates maximum deviation from the average lag, taking into account only the 90% of the total number of samples that are closest to the average.

Figure 7: Lag distribution for just-in-time synchronization as estimated by the orbit model is compared with the actual, measured end-to-end lag. The end-to-end lag of 200 samples was measured. The spread is the maximum deviation from the average lag; the .9-spread indicates maximum deviation from the average lag, taking into account only the 90% of the total number of samples that are closest to the average.

Figure 8: Lag distribution in the combination model for the just-in-time synchronization scheme without the adjustment versus the just-in-time synchronization scheme with the adjustment described in the text. The adjustment makes just-in-time synchronization robust against unpredicted delays.

Figure 9: Curves of all the synchronization schemes as computed by the combination model. Average lag, spread, and .9-spread are given for each scheme.