Reducing Application-Stage Latencies of Interprocess Communication Techniques for Real-Time Interactive Systems

Jan-Philipp Stauffert∗ Florian Niebling† Marc Erich Latoschik‡
University of Würzburg University of Würzburg University of Würzburg

ABSTRACT
Latency jitter is a pressing problem in Virtual Reality (VR) applications. This paper analyzes latency jitter caused by typical interprocess communication (IPC) techniques commonly found in today’s computer systems used for VR. Test programs measure the scalability and latencies for various IPC techniques, where increasing number of threads are performing the same task concurrently. We use four different implementations on a vanilla Linux kernel as well as on a real-time (RT) Linux kernel to further assess if a RT variant of a multiuser multiprocess operating system can prevent latency spikes and how this behavior would apply to different programming languages and IPC techniques. We found that Linux RT can limit the latency jitter at the cost of throughput for certain implementations. Further, coarse grained concurrency should be employed to avoid adding up of scheduler latencies, especially for native system space IPC, while actor systems are found to support a higher degree of concurrency granularity and a higher level of abstraction.

Index Terms: D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming; D.4.8 [Operating Systems]: Performance—Measurements; H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented, and virtual realities

1 INTRODUCTION
Virtual Reality applications often consist of multiple aspects to handle input processing, simulations, artificial intelligence, or rendering etc. To fully take advantage of the available processing power of today’s multicore and/or multi-CPU architectures and to avoid unnecessary blocking, the underlying software routines often will utilize concurrency and asynchronous behavior. At a certain point though, all parts have to cooperate and communicate to generate a consistent world state which usually uses some form of IPC technique. But IPC is heavily affected by scheduler latencies. Scheduler latency is almost unpredictable and might show spikes at uncontrolled points in time, resulting, e.g., in micro stutters. These outliers are presumably not perceived well and may cause increased simulator sickness.

Even without an explicit application concurrency, scheduling impacts the system as the available resources are shared in multiuser multitasking operating system (MMOS) commonly used today. Here, real-time operating systems (RTOS) give promises about an upper bound of scheduler latency. While they are common in, e.g., the embedded world or cyber-physical systems, today’s VR applications usually run on an MMOS (e.g., simply, because there is enhanced graphics acceleration or many existing software solutions). We investigate how different programming languages and IPC techniques behave w.r.t latency jitter when applied on an MMOS compared to an RTOS.

∗ e-mail: jan-philipp.stauffert@uni-wuerzburg.de
† e-mail: florian.niebling@uni-wuerzburg.de
‡ e-mail: marc.latoschik@uni-wuerzburg.de

Figure 1: Schema illustrating the test programs: Each of the n senders is instructed at the same time to get the current timestamp, to send it to a receiver who additionally gets a timestamp and propagates back the elapsed time for logging.

2 PREVIOUS WORK
Frank et al. [3] found visual delay to be a major factor for simulator sickness. Iykovice et al. [4] additionally found latency to influence the performance and experience of test subjects. While they conducted tests with a time invariant latency added, Teather et al. [8] found a reduced performance due to latency spikes. We assume therefore that latency spikes have a similar effect on the experience as degraded latency has. Motion-to-photon latency of VR environments has been measured using different methods, e.g. sine fitting [7]. Several current approaches optimize the rendering stage, e.g., using dynamic time warping or frameless rendering. Our research focuses on a different aspect of the latency problem, namely the latency that occurs prior to rendering, i.e., at the application stage of a VR system.

RT systems guarantee each process to be invoked within a certain time, therefore eliminating spikes in latency for IPC when the receiver has to wait an unbounded timespan until invocation. Most RTOSs are for embedded systems [6] with applications in robotics or industrial controllers. The Linux RT patch is a modification of the Linux Kernel that enables hard realtime capabilities [2]. Improvements in latency often inversely affects throughput. Real-time (getting started as quickly as possible) and real fast (getting done quickly once started) can be considered a design choice [5].

3 METHOD
We implemented a comparable test routine using two distinct IPC techniques and two programming languages: A thread based version using shared memory and mutex locks in C++ and Java and a message based communication with actors using Skala with Akka Actors and the C++ CAF library [1]. The routines start a variable number of pairs of threads/actors and tell the senders to get the current nanosecond time, read an integer from a random location in a memory block of 256k integers and send it to their peer thread/actor. The receiver waits for the hand-over, writes the received integer to another random position back in the memory block, reads the current nanosecond time and logs the difference of the two timestamps (see Figure 1). To ensure that all invoking threads/actors send at the same time, they are synchronized with a barrier. The pseudo-code is described in listing 1 and listing 2. The memory read/write is intended to reliably provoke cache misses for all cases.

The tests were conducted on a system running Ubuntu 14.04 with a Linux Kernel version of 3.14.57 with and without the RT
patch applied in a dual boot configuration. The CPU was an Intel Core™ i7-2700K with 4 cores and hyperthreading. The employed JVM was an OpenJDK 2.5.6 with the Akka 2.3.11 library for Scala Actors. We increased the number of thread/actor pairs running at the same time by the power of two from 1 to 16. For each run, 10,000,000 samples were gathered.

```java
barrier();
t1 = getTime();
x = readMemory(random);
send(t1, x);
```

Listing 1: Sender reads the time and a random memory location and sends it to the receiver.

```java
tl, x = receive()
writeMemory(random, x);
t2 = getTime();
log(t2-t1)
```

Listing 2: Receiver receives the information, saves the variable to a random memory location and calculates how much time has passed.

### 4 RESULTS

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>RT</th>
<th>Mean</th>
<th>Median</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Akka</td>
<td>62.9μs</td>
<td>17.8μs</td>
<td>29μs</td>
<td>4.1μs</td>
<td>6.7μs</td>
</tr>
<tr>
<td>Java</td>
<td>970.3μs</td>
<td>719.9μs</td>
<td>16.9μs</td>
<td>20.8μs</td>
<td>7.4μs</td>
</tr>
<tr>
<td>C++</td>
<td>583μs</td>
<td>329.1μs</td>
<td>13.1μs</td>
<td>29.8μs</td>
<td>3.4μs</td>
</tr>
<tr>
<td>C++ Actor</td>
<td>56.9μs</td>
<td>21.4μs</td>
<td>2.2μs</td>
<td>2.2μs</td>
<td>5μs</td>
</tr>
</tbody>
</table>

Table 1: Comparison values for latencies with and without RT patch with 16 thread/actor pairs.

The latency, however, is better bound with the RTOS, not only for the C++ implementations. Without, there are repeatedly outliers that may deteriorate the user experience and may lead to simulator sickness.

### 5 CONCLUSION

Linux RT reduces latency jitter at the cost of some overall performance in the C++ case, an acceptable trade-off for VR systems. Additionally, system space IPC concurrency should be limited to a certain extent of granularity to reduce the impact of scheduler latency. Running on an RTOS, the Actor model provides a valuable alternative for an increased degree of concurrency granularity, specifically using the C++ implementation. Still, with our implementation based on the Java VM, latency spikes could not be lowered so far as their cause is not the system scheduler but the garbage collector (GC). Future work will evaluate if different GC implementations and parametrization can alleviate this problem.

Overall, while VR applications need concurrency and modularity to handle all required software tasks, communication can induce problems if proper care is not taken and adequate performance measures are not performed frequently as a standard procedure. We have only looked at a basic n × 1 : 1 IPC but see the need to extend the research to test the impact of different approaches as well as to extend the technical analysis with user-based perception studies to relate technical measures to perceived qualities, e.g., to see if and how it makes sense to trade performance for a bounded latency.

### REFERENCES