ABSTRACT
Latency is a pressing problem in Virtual Reality (VR) applications. Low latencies are required for VR to reduce perceptual artifacts and cyber sickness. Additionally, latency jitter denotes the variance in the pattern of latency changes which additionally may cause unwanted effects. This paper analyzes latency jitter caused by typical inter-thread communication (ITC) techniques commonly used in todays computer systems employed for VR, the influence of the operating system scheduler, and the effect of different garbage collection (GC) methods to understand their effect on latency spikes, here for different Java Virtual Machines (JVM). We measure the scalability and latencies for various ITC techniques with an increasing number of threads and actors performing prototypical concurrent tasks. Four different benchmark implementations on a vanilla Linux kernel as well as on a real-time (RT) Linux kernel 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 ITC techniques.

We confirmed that scheduler and prioritization of the VR application both play an important role and identified the impact they have on the implementation strategies. Also, Linux RT can limit the latency jitter at the cost of throughput for certain implementations. As expected, the choice of a GC method also is critical and will change the latency patterns drastically. As a result, we suggest that coarse grained concurrency should be employed to avoid adding up of scheduler latencies and unwanted latency jitter for the native ITC case, 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 components to handle input processing, simulations, artificial intelligence, or rendering etc. Non-functional software quality requirements like modularity, maintainability, and reusability can have an unforeseeable impact on the temporal behavior of software, especially for a Real-Time Interactive System (RIS), i.e., in Virtual, Augmented, and Mixed Reality (VR, AR, and MR) and computer games. Due to the complexity of many RIS applications, they often split into different parts to foster cohesion and decoupling. To exploit todays’ multi-core and multi-CPU architectures and to avoid unnecessary blocking, these parts often will be executed concurrently or they will be completely distributed [2, 15].

At a certain point though, all parts have to cooperate and communicate to generate a consistent world state which implies some sort of inter-process communication (IPC) or inter-thread communication (ITC). Hence it is critical to understand the impact of IPC/ITC on the resulting latency patterns. IPC/ITC 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. As a result, latency and latency jitter can severely disturb the performance and experience of users and it can cause 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. We investigate how different programming languages and ITC techniques behave w.r.t latency and latency jitter when applied to (a) an MMOS compared to (b) an RTOS. Three languages will be used: (1) C++ to create native binaries and (2) Java and Scala targeting the JVM. We finally compare threading with mutex locks to an actor model implementation.

2 RELATED WORK
An early discussion of simulator sickness is led by McCauley et al [17], Frank et al. [9] found visual delay to be a major factor for simulator sickness. Ivtovic et al. [12] 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. [22] 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. Users might be able to compensate for a predictable overall latency when it comes to interaction tasks (not for the perception though) but they can’t compensate for unpredictable latency spikes.

Recent work has been done to reduce motion-to-photon latency of VR environments. Here, the overall goal is to apply the most current sensor reading as late as possible in the final graphics rendering stage to avoid, e.g., the application of an outdated camera projection. This latency cause has been measured using different methods, e.g. sine fitting [21]. Several current approaches optimize the rendering stage, e.g., using dynamic time warping or frameless rendering, light sensing [5], and automated frame counting [10]. Our research focuses on a different source for 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 ITC when the receiver has to wait an unbounded timespan until invocation. Most RTOSs are for embedded systems [20] with applications in robotics or industrial controllers. The Linux RT-Preempt patch modifies the Linux kernel to enable hard realtime capabilities [6]. This allows the comparison of software performance under both operating system flavors. Other operating systems exist only in either MMOS or RTOS variant. 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 [18].
3 Method

We will examine differences in ITC latency jitter on a MMOS and RTOS to evaluate whether an RTOS can prevent critical latency spikes and how different programming languages and ITC concepts need to be adapted. We distinguish between two platforms: Native binaries are compared to bytecode running on the JVM. Additionally, the traditional multithreading approach with threads and mutexes is compared to the abstraction of actors.

Threads are used for concurrent flows of execution inside of one process. Here, mutexes allow threads mutually exclusive access to a resource with threads waiting for a resource being able to yield their execution time to another thread or process. Mutexes therefore allow for better real-time behavior than spinlocks that poll for a resource to be available [6]. Actors on the other hand provide an abstraction to facilitate parallel programming usually based on threads and lock-free communication as used for VR applications in [14]. Actors are entities that run in parallel and which solely communicate by message passing [13].

3.1 Test Routines

We implemented a comparable test routine using two distinct ITC techniques and two programming languages:

1. thread based using shared memory and mutex locks in C++
2. thread based using shared memory and mutex locks in Java
3. message based with actors using C++ with the CAF library [4]
4. message based with actors using Scala with Akka [16]

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 handshake, 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. In larger applications, cache misses will certainly occur due to the increased code and large assets, therefore urging the processor to load data from the slower main memory instead of the much faster caches. All tests shown here collect 10,000,000 samples. Latency jitter is introduced by, among others, the OS scheduler, other processes and hardware interrupts that delay the communication.

The actor implementations use default settings. Akka uses by default a fork-join thread pool with work stealing and three times the amount of threads than processors with a central coordinating scheduler [4].

The C++ Actor Framework uses by default a thread pool with the same amount of threads than processors with a central coordinating scheduler [4].

3.2 Hardware

The tests were conducted on a computer running Ubuntu 14.04.3 with a Linux Kernel version of 3.14.57 with and without the RT-Preempt patch applied in a dual boot configuration. The CPU was an Intel® Core i7-2700K with 4 cores and hyperthreading. The employed JVMs were an OpenJDK 2.6.3 with the Akka 2.3.11 library for Scala Actors and for GC comparison additionally the Zing JVM version 1.8.0-zing_15.09.0.0-b6.

3.3 Schedulers

Linux supports multiple schedulers with different use cases [3]. If no special scheduler is requested, Linux defaults to the “other” scheduler (SCHED_OTHER). Real-time scheduling is done with SCHED_FIFO, which implements a FIFO principle, SCHED_RR, a round-robin approach, or recently SCHED_DEADLINE, which executes the thread with the earliest deadline first.

The C++ implementations will use the round-robin scheduler for it shows the best results in terms of limiting latency jitter for our use case. The implementations running on the JVM are evaluated both for the SCHED_OTHER and the SCHED_RR as they are impacted differently by the choice of the scheduler. SCHED_DEADLINE will be evaluated in later work.

The test programs are run with the round robin scheduler at priority 90, which is above most other processes with the exception of certain kernel processes like “watchdog” and “migration” that are essential for the proper functioning of the OS. When using the default scheduler, no further prioritization like nice values are used.

Hardware interrupts will nonetheless be served immediately urging other processes to wait. With threaded interrupts this time is held as short as possible with a big part then taken care of in a kernel thread that is subject to the scheduler [7].

The choice for a scheduler is a sensitive one. It should be evaluated which one performs best for the software at hand. While our test implementation with Scala/Akka performs better in terms of latency jitter with the default scheduler as shown below, it doesn’t make any promises or efforts for real-time behavior and should not be favored for RT scenarios.

3.4 Vanilla vs. RT-Preempt

All four implementations were run on Linux with and without the RT-Preempt patch applied using 16 threads or actor pairs. Figure 2 depicts the latency for each sample that took more than the median + 2 * standard deviation. Table 1 shows the absolute performance values for comparison.

The C++ implementations are heavily affected by scheduler latency on the MMOS Linux and drastically reduce latency spikes using Linux RT. The native thread implementation sees a decrease

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

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

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

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

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.
in average performance with the RT patch but shows fewer outliers that are bound to a lower maximum latency. The C++ Actor implementation benefits from the reduced maximum latency while not suffering the same performance degradation as the native threads. However, there are spikes in the beginning of the test run on the RTOS which are not present on the MMOS. This hints to the actor initialization having more impact there.

3.5 Garbage Collectors

Preliminary studies with a modified version of the here presented tests showed the GC as a major impacting factor for latency jitter.

The Java Garbage Collection is as beneficial for the language as it poses problems. It allows for reliable software as a consequence of the nonexistence of memory corruptions and memory leaks. The well known problems of the garbage collection are temporary program stalls while the GC is conducting its work. Many application areas don’t mind pauses in execution under 1 second. Due to Java’s ubiquity especially in the business world, it has advanced to areas where latency plays a crucial role for the business value like in high frequency trading [19].

For our tests, we use the four different GCs that are implemented in the OpenJDK, which are the Serial, Parallel, Concurrent Mark Sweep (CMS) and G1 GCs. Additionally, the Zing JVM [23], which promises pause free GC, is examined.

Figure 3 shows the measurements for the Scala/Akka implementation with Figure 4 showing the respective measurements for the Java thread implementation. The time was normalized to the range \([0; 1]\) but different settings led to differing run times. With the exception of the Zing GC, every test has information added where the garbage collection took place. Java garbage collection is divided into two different steps, the Young generation (YG) GC and the Old generation (OG) GC, where short living objects are faster to
Figure 3: Comparison of different GCs for the Scala/Akka implementation using 16 actors with the default Linux scheduler (SCHED\_OTHER) and the round robin scheduler (SCHED\_RR) running on a Linux with and without the RT patch applied. The y-axis is differently scaled for each test to better convey the individual behaviour that would otherwise get lost due to the big difference in latency behaviour.

<table>
<thead>
<tr>
<th>GC</th>
<th>Scheduler</th>
<th>Max</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RT</td>
<td>Vanilla</td>
<td>RT</td>
<td>Vanilla</td>
</tr>
<tr>
<td>CMS</td>
<td>SCHED_OTHER</td>
<td>67,ms</td>
<td>63.4,ms</td>
<td>1.9,ms</td>
</tr>
<tr>
<td></td>
<td>SCHED_RR</td>
<td>191,ms</td>
<td>212,ms</td>
<td>18.3,ms</td>
</tr>
<tr>
<td>G1</td>
<td>SCHED_OTHER</td>
<td>241,ms</td>
<td>216.8,ms</td>
<td>3.4,ms</td>
</tr>
<tr>
<td></td>
<td>SCHED_RR</td>
<td>235.9,ms</td>
<td>210.6,ms</td>
<td>45.1,ms</td>
</tr>
<tr>
<td>Parallel</td>
<td>SCHED_OTHER</td>
<td>91,ms</td>
<td>110.2,ms</td>
<td>3.1,ms</td>
</tr>
<tr>
<td></td>
<td>SCHED_RR</td>
<td>375.6,ms</td>
<td>192,ms</td>
<td>212.7,ms</td>
</tr>
<tr>
<td>Serial</td>
<td>SCHED_OTHER</td>
<td>631.8,ms</td>
<td>385.1,ms</td>
<td>4.2,ms</td>
</tr>
<tr>
<td></td>
<td>SCHED_RR</td>
<td>1423.5,ms</td>
<td>717.4,ms</td>
<td>90.8,ms</td>
</tr>
<tr>
<td>Zing</td>
<td>SCHED_OTHER</td>
<td>744.3,ms</td>
<td>397.5,ms</td>
<td>11.2,ms</td>
</tr>
<tr>
<td></td>
<td>SCHED_RR</td>
<td>720.1,ms</td>
<td>655.6,ms</td>
<td>137.7,ms</td>
</tr>
</tbody>
</table>

Table 2: Comparison values for latencies for the Scala/Akka implementation running with different GCs. The tests were conducted with and without RT patch with 16 actor pairs.

get collected. Longer living objects and those surviving the YG GC are cleaned up less often but with more impact on the system. As the test routines are only shortlived and only create objects to pass the results around, they mostly don’t provoke a Full Garbage Collection that sweeps the OC. Depending on the GC, this can happen nonetheless as is seen for the Serial and Parallel GC. There, spikes in latency are the result of the longer running full GC. If full GC is provoked for the other GCs, the impact is comparable.

The runtime and frequency of the full GC can be adjusted with the assignment of different ratios of the available memory to the YG and OG. This can be done to trade less frequent garbage collection for an increased stall time of the program but doesn’t change the phenomenon of programm pauses in general. While this makes it possible to delay garbage collection in our tests until after the test run, we kept the memory assignment with default values to have our tests exhibit common behaviour.

Furthermore, the memory usage was not optimized for this case. It is possible to work around the GC by allocating memory without
its knowledge that has then to be managed manually. Additionally, the GC’s work can be alleviated by reusing objects. Since these measures are not used in common Java programming, they are omitted for the comparison here.

The tests were run with both the FIFO and the round robin scheduler because the decision for the SCHED
RR is not as obvious as in the C++ tests.

4 Results
Both JVM-based implementations show repeated spikes, apparently caused by garbage collection. Here, Akka spikes are worse, as the message system seems to suffer under the many messages that are sent to log each result, therefore creating a lot of short-living objects. The values in Table 1 and Figure 2 show selected combinations of GC and scheduler (Scala/Akka: Concurrent Mark Sweep GC, SCHED_OTHER; Java: Concurrent Mark Sweep GC, SCHED_RR). The Akka measurements show far longer mean latencies which is a result of an implementation problem. The increasing delay occurred when changing the test program to not only send the measured latency but also the time when it occurred for logging reasons. The reason for this has to be assessed in the future. This problem makes it not possible to compare the Akka version to the other implementations but only to itself. Preliminary studies showed Akka’s performance comparable or better than Java’s but were not yet replicated with different GCs.

Both actor implementations show a better scaling with more parallelism. More threads lead to more mean and maximum latency, while the actor values are growing at a slower pace.

The Scala/Akka implementation shows more and larger latency spikes with the SCHED_RR which should support RT, but profits from the changes to Linux with the RT-Preempt patch if no RT scheduler is used. All test runs show outliers, which can be partially explained with full GC for the Serial and Parallel GC. The Java implementation has similar patterns for the Serial and Parallel GC, where repeated spikes are provoked by the full GC but besides this the latencies stay bound. The other GCs show very specific patterns with the CMS GC exhibiting repeated spikes, the G1 GC starting with large latencies that go down to then slowly increase again and the Zing GC that has times with higher latencies around certain points in time though still very low latencies in comparison to other GCs.

We tried to introduce pauses of 1ms after each measurement to see if the OS and GC can make use of this time window of inactivity. The Scala/Akka implementation reduced its outliers roughly by a factor of four, which is probably due to skipping certain internal mechanics that cause latency spikes. It is more a working around spikes by hoping that certain characteristics fall into the pause than a real solution. The C++ implementation showed a slight increase in latency, which is explained by the added overhead of waking up the main application that then invokes all the threads that have been
idle in the mean time.

While we tried CPU pinning for the thread implementations, it didn’t affect the measurements. CPU pinning describes the process of assigning a thread to a CPU. If this is not done, the scheduler is free to schedule a thread each time it is resumed on a different CPU. If a thread changes the CPU, the cache of the new CPU might not have pre-loaded the required data and additional time is wasted to load the program code and data. This is most likely due to our implementation being too small in terms of executable size to make a difference. Other use cases e.g. in big data applications found a significant improvement of throughput by assigning threads to CPUs [8].

5 Discussion

The threaded RTOS versions exhibit an increased mean and maximum latency with increasing number of threads, eventually surpassing the latency of outliers that happen with the MMOS versions. Therefore, a very fine-grained concurrency using many OS threads should be limited or an actor-based system should be applied. An actor-system implemented in user-space can make better use of the application’s concurrency without burdening the OS scheduler with huge amounts of threads. We find the RTOS performing worse in the mean and median while also scaling worse with more threads. The scaling is negatively affected by the additional scheduling complexity in Linux RT where mutexes in the kernel lead to more context switches and therefore more overhead. 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.

Applications can change their behaviour in different environments. Therefore, it is important to test every application in different settings to determine the best configuration. Here, the C++ Actors have an initialization cost under Linux RT and should therefore be created in program sections where latency outliers have lesser impact. With the unpatched Linux, this adaptation is not needed. Even more varying behaviour is shown by our Akka test, which performs better either on the patched or unpatched Linux depending on the scheduler and garbage-collector.

Furthermore, the JVM does a lot of optimizations behind the scenes [11] that we did not fully account for that can lead to different behaviour in other scenarios. The test applications presented here are small and run fast to allow quick testing of various settings. Larger applications might be handled differently and get more and more optimized the longer they are running. In the financial sector, some JVMs are “warmed up” for hours before they are put in production [1].

Only 1:1 communication with a certain amount of threads/actors in parallel at the same time was investigated. In a VR application, the communication might not be in sync. Input devices will report their measurements at different times and their messages are propagated through the program components in different paths. The amount of entities taking part in a communication will vary. This research is trying to start the evaluation of latency jitter sources for a restricted communication pattern, which will be present in more complicated settings as well.

The analysis only sheds light onto a very selected piece of VR applications. Even in this restricted test setting, there are many influencing factors that can provoke latency spikes that can only partially be avoided. Hardware/firmware interrupts can influence the CPU in a way that is not preventable from an application side. Bigger applications will suffer from even more sources of latency due to the underlying hardware and operating system, which makes controlling latency jitter even more difficult.

6 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 ITC 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 GC. Different GCs provoke special latency patterns that need to be found out for every application anew and be considered. As long as a language running on the JVM is the choice for a VR project, the GC has to be accounted for.

Overall, while VR applications need concurrency and modularity to handle all required 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 \( n \times (1:1) \) ITC 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 lower latency spikes.

There is a need for special hardware and software to avoid latency jitter and provide an immersive experience. The VR community has identified the problem of latency and is working on it. We want to stress that low mean latency is not enough but the jitter has to be kept low as well.

References


