A Framework for Wait-Free Data Exchange in Massively Threaded VR Systems

Patrick Lange, Rene Weller, Gabriel Zachmann
University of Bremen
{lange,weller,zach}@cs.uni-bremen.de

ABSTRACT

A central part of virtual reality systems and game engines is the generation, management and distribution of all relevant world states. In modern interactive graphic software systems usually many independent software components need to communicate and exchange data. Standard approaches suffer the n^2 problem because the number of interfaces grows quadratically with the number of component functionalities. Such many-to-many architectures quickly become unmaintainable, not to mention latencies of standard concurrency control mechanisms. We present a novel method to manage concurrent multithreaded access to shared data in virtual environments. Our highly efficient low-latency and lightweight architecture is based on a new *wait-free* hash map using key-value pairs. This allows us to reduce the traditional many-to-many problem to a simple many-to-one approach. Our results show that our framework outperforms by more than an order of magnitude standard *lock-based* but also modern *lock-free* methods significantly.

Keywords

Concurrent data structures, parallel programming, memory management, progress guarantee, map, dictionary

1 INTRODUCTION

Modern virtual environments (VEs) usually consist of many different components such as graphics rendering, sound, several input devices, haptic rendering, physically-based simulation, etc. A similar situation exists in many modern games. All these components have to share and communicate some kind of data. For instance, the physically-based simulation gathers data from the input devices and passes its results to the graphics, haptic and sound rendering. This requires some kind of interface for the data exchange between the components. That data can be extremely large, e.g. think about a spacecraft simulation in the asteroid belt, where the position of thousands of asteroids changes continuously. All transformations have to be passed from the simulation to the rendering component.

One of the standard approaches to describe and encode virtual environments is the classic fields-and-routes-based data flow paradigm, as for instance specified in VRML and X3D. In that paradigm, we have to draw "wires" between the in- and output fields of the components and route the data through these wires.

Consequently, the number of interfaces grows quadratically with the number of components. This is manageable, as long as the number of components is relatively small. However, modern virtual environments are often not restricted to a single user but may contain thousands of users that are distributed over the whole world and connected via a network. Additionally, adding new components to the system requires changes to the interfaces of many other components and the drawing of new routes. This reduces the maintainability significantly and can affect the performance of the overall system negatively.

A second major challenge for modern VR systems is the data consistency: for instance, in multi-user VEs, all users interact simultaneously but the system has to provide a consistent view on the VE to all users. This problem also arises in multithreaded single-user VEs, for instance if a haptic rendering thread runs at 1000 Hz and the graphical rendering requires only 30Hz. As a result, we need some method to synchronize the data exchange between the components. Classic VR systems often use locks on shared data to avoid race conditions and barriers for data synchronization. Unfortunately, this decreases the performance of the whole system because components have to wait until all other components have finished their operations. Consequently, it is extremely complicated to guarantee time-critical access to data for the components. In case of haptics this may result in poor immersion or even damage of the expensive devices.

Both challenges, the distribution and the synchronization of the data, are closely related. However, classic VR systems usually handle them separately. In this paper we present a new database-based approach that solves both problems simultaneously and enables us to overcome all the limitations.

In detail, our contributions in this paper are

- an easily adaptable and maintainable software architecture for very large, massively parallel VEs,
- a novel, wait-free underlying mechanism to facilitate very efficient access to shared data,
- significantly reduced, overall synchronization overhead.

The basic idea is very simple: first, we identify all kinds of data that need to be shared between different components, e.g. the transformations of the objects in the scene. Instead of drawing a quadratic number of routes between the components, we assign a single unique *key-value pair* to each data packet. We register this key to a *global database* and reserve memory for the data.

This global database holds the complete shared *world state* of the VE. If any component wants to access the data, it simply has to look up the *key* in the database. Note that we actually do not require a full-fledged database with SQL-like access, but a simple dictionary, implemented as a hash-map, fulfills all our requirements. Basically, we differentiate between *consumers* that just *read* the data, e.g. the rendering thread that reads the transformations of the 3D objects in the scene, and *producers* that are allowed to *write* data, like the physically-based simulation that changes the transformations. A major advantage is that our global database automatically reduces the many-to-many interface of classic approaches to a simple many-to-one interface. Additionally, this reduces the synchronization overhead (see Figure 1).

In order to guarantee a wait-free write and read access, we propose a double-buffering scheme for the writing operations in our approach. Obviously, several consumers can read the same key-value pair in parallel, while one producer is allowed to change the key-value pair. This avoids the waiting-times, known from classical approaches using e.g. barriers. Moreover, adding new interfaces between existing components of the software system or adding new components to the system, is extremely simple and does not necessarily require the introduction and implementation of more interfaces between components: we only have to introduce a new key-value pair to the database.

Overall, we get a highly adaptable wait-free system for massively-parallel access in large VEs. In the following we will describe our implementation of such a database-based VR system and we will present results of timings that show that our new system outperforms classic approaches significantly.

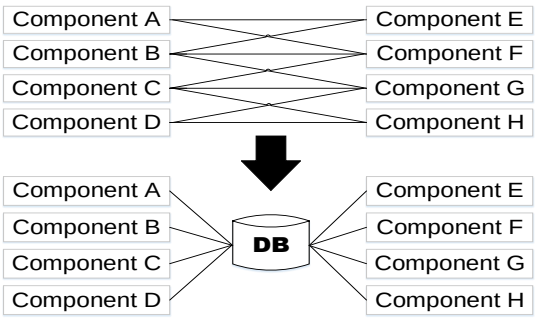

Figure 1: Comparison of our approach (bottom) to the standard approach (top). While the standard fields-and-routes-based data-flow for data exchange and communication incurs a quadratic number of communication interfaces, our approach reduces this significantly. Consequently, it improves system performance and maintainability.

2 RELATED WORK

A complete overview of all data organization methods in VR systems would exceed the scope of this section. Hence, we will concentrate on modern system designs that rely heavily on multithreaded access to shared system resources and we will avoid classic methods that where primarily designed for sequential access.

When introducing such a *global database* we will immediately encounter the well-known problem of concurrent data structures and race conditions.

A basic distinction feature for concurrent data structures is whether they are *blocking* or *non-blocking*. *Blocking* approaches usually allocate resources exclusively by using various well-studied techniques such as mutexes, semaphores or condition variables. Concurrent threads have to wait until a resource has been released. This may result in a loss of efficiency and parallelization or even deadlocks. Many scenegraph systems like OpenSG use blocking mechanisms for synchronization. *Non-blocking* approaches avoid this exclusive allocation of resources by introducing very smart designs or by allowing changes only in a very small critical section using atomic operations. These atomic operations, like *Compare-and-Swap (CAS)*, are usually directly supported by the processor. Consequently, *non-blocking* data structures avoid inconsistencies and deadlocks.

Today, there exist efficient non-blocking implementations for almost any common data structure [Her91], [Her05], [Fel13]. However, due to the restriction to processor-supported primitive data types that allow atomic operations, they can be hardly extended to VEs that require more complex data structures, e.g. matrices to store transformations. Additionally, memory management has to be taken into account: the design has to ensure that under no circumstances memory is freed, which is still in use by a concurrent thread.

Actually, *non-blocking* approaches can be further classified into *lock-free* and *wait-free* methods. *Lock-free* approaches do not use any locks and guarantee progress of at least one of the threads accessing the shared data structure. Lock-free approaches incorporate that some threads can be delayed arbitrarily, in most cases the producer, which waits until every reader (which is in most cases wait-free) has finished its operations on the shared data structure. This approach is vulnerable for deadlocking the producer, however, statistically all threads will make progress [Her91]. *Wait-free* approaches guarantee each thread access to the shared data structure in a finite number of steps, regardless of other threads accessing the shared data structure [Her91].

Wait-free approaches have been developed for queues [Stel09] or linked lists [Tim12]. Some lock-free solutions have also addressed this challenge, including per-thread timestamps [Bra13], [Har01], reference counters [Gid09], expensive compare-and-swap (CAS) approaches [Det01] or global pointers, such as [Her05] and [Mic04]. They only support primitive data types and they are not suited for random-access, which makes them unsuitable for complex system architectures for VEs. The main problem is that there are no atomic operations on hash maps available, which retrieve the position of a given key inside a hash map and replace or return the found value.

With regard to the overhead needed for synchronizing the access of threads, wait-free approaches offer the least overhead, while lock-free mechanisms incur more overhead, and lock-based mechanism even more. Wait-free approaches additionally support different thread cycle times because no thread is blocked by another, promising high performance when accessing a shared data structure. Some of the above stated approaches had been compared by Hart et al [Har07]. Hart summarizes, that the reclamation overhead of non-blocking schemes can dominate the overall execution time of these approaches, decreasing the performance boost with respect to traditional blocking approaches. Hart also concludes, that for accessing single data sets the *hazard pointer* scheme performs very well, except when these data sets have to be traversed as the scheme uses atomic instructions. He further concludes, that it is desirable to create a *hazard pointer* based scheme that avoids per-element atomic instructions. Our approach takes this into account and enhances the *hazard pointer* scheme, which was previously introduced as a lock-free approach for hash maps [Mic04]. Our scheme avoids atomic per-element instructions and improves the management of thread-local *hazard pointers* by both wrapping the access of the hash map and by defining the key-value pairs as tuples with a special *copy-on-write* mechanism.

3 OUR APPROACH

In this section we present our new wait-free hash map that allows fast concurrent access for both read- and write-operations. First, we will describe the basic concepts of our *key-value database*. Then we will give an overview on our implementation.

3.1 Basic Concept

The core of our data structure is the *database*. It stores all data that can be accessed concurrently, either by the consumer or producer components that we summarize as *entities*. The shared data is stored in the *database* in form of *key-value* pairs. Basically, the *key* of such a key-value pair is the identifier for the entities. There is no global main loop required; each entity can access the data, i.e. read or write, at any point in time.

The main challenge is to avoid inconsistencies, as the database has to ensure that no data that is currently read by an entity will be overwritten by another entity that concurrently writes the data. Moreover, all these access operations should be performed without the necessity to wait for any entity.

In order to guarantee the consistency of the data, we propose a *copy-on-write* mechanism: if a producer asks the database for writing access to some key-value pair, we simply update the original data and deliver a copy of this original data to all consumer entities of the key-value pair. All entities with reading access are not disturbed, they can still read their old data. When the producer has finished its writing operation, we simply deliver this new data to future entity queries for this key-value pair. We call data that is currently written by a producer the *working reference*. Data that is accessed by entities for reading is called *working copy*. The overall value is therefore defined as a pair, with the first element being the working reference and the second being the working copy.

Basically, this mechanism is somewhat similar to double-buffering, known from graphical rendering. In contrast to that, we have to ensure that old working copies of the key-value pairs will be deleted if they are not accessed by any entity anymore. In order to realize this, we adopted the concept of *hazard pointers*.

If an entity asks for access to a key-value pair, the database generates a hazard pointer that indicates that this data is currently being read by an entity. The hazard pointer will be released when the entity has finished the access. All hazard pointers of all entities are managed globally by the database. After releasing a hazard pointer, the database checks if there are other entities still reading the same data. If this is not the case, and there already exists a newer copy of the data that had been produced meanwhile by a producer, the old data can be safely freed.

In contrast to our approach, the original lock-free scheme of Michael [Mic04] allows multiple producers, but it only supports primitive data types. It also requires the use of atomic instructions when updating values in the hash map, which reduces the performance significantly (see Section 5). Our copy-on-write mechanism avoids these drawbacks and eliminates the expensive atomic instructions. Additionally, we support complex arbitrary data structures like lists, matrices, dynamic arrays or vectors in the key-value pairs. Currently, the price for these advantages is that only one producer for each key-value pair with an arbitrary number of consumers is supported.

3.2 Overview of the Implementation

Even if the basic concept is relatively simple, its actual implementation holds some challenges. In the following, we will describe these challenges and present solutions. Figure 2 illustrates the basic concept of our architecture and shows the main components as software classes.

In the previous section we described the three main components of our approach: the database, the key-value pairs and the entities. Our implementation contains a class for each of these components.

Figure 2: Overall system architecture, constituted by the global database named *KeyValueDatabase*, the entities named *SimulationEntity* and the key-value pairs named *KVDataPair*.

3.3 The KeyValueDatabase Class

The global database has to provide three core functionalities:

- Putting values into the database
- Getting values from the database
- Release of unused memory

The `putDB` function is used to insert a key-value pair into the database. If the key is not already stored in the database, it simply creates a new key-value pair. Otherwise the existing key-value pair will be updated. The value can be retrieved in constant time using a simple hash function. Moreover, the `putDB` function contains the proposed copy-on-write mechanism: when we update the value for a key, the *working clone* of the value is updated by a clone of the *working reference*. Note that the memory of the old working clone is untouched and no memory is freed at this point, only the pointer has been replaced.

The `getDB` function is used to retrieve an existing key-value pair from the database. However, we have to indicate whether we have to return the working reference or the working clone. To do that, we additionally include a variable that indicates the access right.

Finally, the database implements the `scan` function to free allocated memory, based on Michaels approach [Mic04]. The `scan` function checks whether the entries of a given list of retired `KVDataPairs` (retired key-value pairs, which are returned by a put function call) are currently under use. More specifically, if there is any hazard pointer from a concurrent thread pointing towards a given entry. The memory of the key-value pair will be freed if there is no intersection between the list of a threads retired `KVDataPairs` and the global hazard pointer list (see Michael [Mic04] for more details).

Algorithm 1 Database putDB(key,yalue)

```
if key in map then
    pair(KVDataPair) slot = map.getValue(key)
    slot.workingreference = value
    KVDataPair retired = slot.workingclone
    return retired
else
    map.insert(tuple(key,value))
```

Algorithm 2 Database getDB(key,access)

```
if key not in map then
    return empty
else
    pair(KVDataPair) slot = map.getValue(key)
    if access is producer then
        return slot.workingreference
    else
        return slot.workingclone
```

3.4 The SimulationEntity Class

Each software component that can access the central database is considered as a thread with a main loop running within an execute function. Inside the main loop, the software component can access the central database. For arbitrary purposes we define an abstract class called `SimulationEntity`. The class serves as a wrapper for every software component that wants access to the central database. It provides two wrapper functions of the `put` and `get` function. These functions additionally provide the management of the thread-local lists of acquired hazard pointer as well as retired working clones.

The `putE` wrapper calls the `putDB` function of the central database and retrieves the retired `KVDataPair`, if available. The retired `KVDataPair` is inserted into a thread-local list of retired `KVDataPairs`, which is later used to free the retired working copy.

Similar to the `getE` function of the central database, the `getDB` wrapper function distinguishes whether the access is from the producer or from a consumer of the key-value pair. If the producer of the key-value pair access the database, the corresponding key-value pair (replaced working clone) is returned. As there is only one producer for each key-value pair no memory management is needed because the producer works on the working reference of the database record. If a consumer wants to access the database, a hazard pointer is created for the record and saved to a thread-local list of used hazard pointers, indicating that no other concurrent thread should free the memory of the key-value pair.

The calling of several `putE` and `getE` inside the main loop of a thread, inserts arbitrary `KVDataPairs` (old working clones of threads, produced by updating key-value pairs) and hazard pointer (references to used key-value pairs) to the thread-local lists. After finishing all operations in one thread cycle, the `release` function is called at the end of the thread's main loop. The `release` function releases all acquired hazard pointers from `getE` calls, indicating other threads that the memory can be safely freed. For producers that additionally call the `putE` function, the second part of the release function tries to free old working clones of the thread produced key-value pairs, by calling the scan function of the central database. This is similar to the deletion of retired pairs from the `KVDataPair` list.

At the end of the main loop of each entity all used references to key-value pairs are released. It may happen that an arbitrary number of old working clones could not be freed because some concurrent threads are still using them. Due to the *wait-free* access of the central database that guarantees progress of each thread, every claimed memory of a key-value pair will be freed after some time. The maximum time is defined by the thread cycle time of the slowest key-value pair consumer.

Algorithm 3 SimulationEntity putE(key,value)

```
KVDataPair retired = database.putDB(key,value)
if retired is not null then
    retiredKVDataPairs.add(retired)
```

Algorithm 4 SimulationEntity getE(key,access)

```
if access is consumer then
    KVDataPair value = database.getDB(key,consumer)
    HazardPointer hp = value
    acquiredHazardPointers.add(hp)
    return value
return database.getDB(key,producer)
```

Algorithm 5 SimulationEntity release()

```
for all acquired hazard pointers hp of entity do
    release hp
database.scan(GlobalHazardPointers, RetiredPairs)
```

3.5 The KVDataPair Class

A key-value pair is represented by an abstract class named `KVDataPair`. Each `SimulationEntity` has to define its own database records by defining member variables. These member variables can be accessed via a key, which is determined when the key-value pair is first stored in the database. This allows us to generate an arbitrary number of complex values for each key and it reduces the amount of required key-value pairs. Moreover, it avoids unnecessary calls of `getE` functions.

4 CASE STUDY

We applied our approach to a typically highly parallelized VE environment for simulated space missions. More precisely, we adopted our system to a simplified version of ESAs ARCHEO-E2E system [Neg12] that defines a reference architecture for spacecraft engineering feasibility studies. Instruments of the spacecraft, as well as the environment, including the spacecraft's orbit and attitude, are simulated and defined as entities within the software architecture. The sensor input for the instruments is synthesized from the simulated environment. In our implementation, all this synthesized data and the current world state (e.g. spacecraft pose, positions of celestial bodies, sensor configurations, scene nodes) are represented as key-value pairs in our central database. The instruments and the physically-based simulation read and write the entries periodically. Consequently, this scenario has a large amount of concurrent read- and write operations on our database. Figure 3 illustrates the current simplified architecture.

Figure 4 shows a rendering of the simulation in which a spacecraft travels through the main asteroid belt of our solar system. You can see a simulated spacecraft probe, while approaching a target asteroid for scientific experiments.

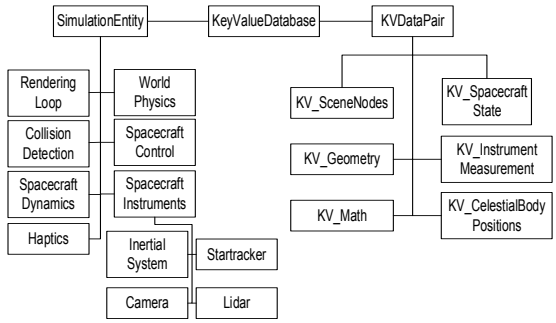

Figure 3: Case study: Simplified architecture of an end-to-end space mission simulator.

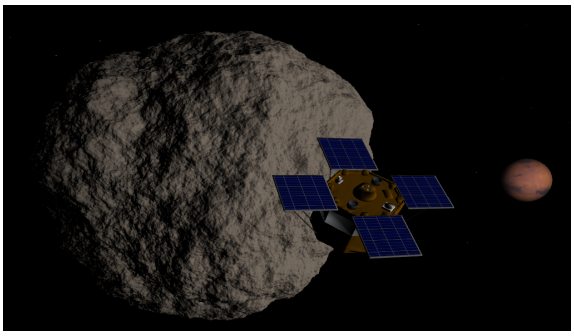

Figure 4: Case study: Simulated spacecraft rendezvous with target asteroid.

5 RESULTS

We run our experiments on a machine with an Intel Core i7 4-core processor with enabled Hyperthreading, operated by Windows 7 64 bit and 4GB of RAM. Due to the limitations of the competitive approaches we had to limit the data pairs to basic primitives like doubles, integers and pre-allocated lists with fixed size of 256 Bytes, in order to obtain a fair comparison with our novel approach. We performed 50,000 read- and write-operations for each test. Additionally, we repeated each individual test 100 times and averaged the resulting timings. We performed the test for different numbers of threads ranging from 2 to 128.

In our *wait-free* key-value database test runs, we inserted the keys for each test run at random positions in order to prevent caching. The key size was set to 10 Bytes.

We compared the performance of our new approach to two different existing methods. The first competitor was a standard *blocking* hash map. We used the well-known boost library that uses shared mutexes and allows multiple readers and a single writer accessing the complete hash map. Additionally, we adopted a *lock-free* hash-map of the original hazard-pointer algorithm that supports *wait-free* reading and *lock-free* writing from [Mic04].

Figure 5 and Figure 6 show a comparison of the performance. You can see that our key-value database outperforms both competitors for reading as well as writing operations.

More precisely, our method is more than an order of magnitude faster than the traditional lock-based hash map for reading operations. Obviously, the speedup increases with an increasing number of threads because the concurrent thread access is limited by the locks in the standard approach (see Figure 5). However, the data cloning for writing operations in our approach takes some time. Hence, if the number of concurrent threads is small (<8), the standard locking method could be slightly faster than our approach (See Figure 4). For more than 8 threads, our approach outperforms the *lock-based* approach significantly by a factor of two.

Even more interesting is the comparison with the *lock-free* hash map. Our new *wait-free* method, as well as the *lock-free* method, supports *wait-free* concurrent reading access of the data. Consequently, both methods perform almost identically for reading operations (see Figure 5). However, our method also allows *wait-free* writing access using the copy-on-write mechanism instead of CAS, used by the *lock-free* hash map. In that case our method outperforms the *lock-free* competitor by a factor of up to 7, depending on the number of threads.

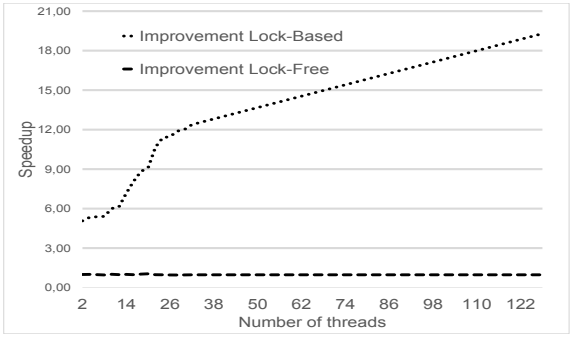

Figure 5: Speedup compared to our approach for read operations.

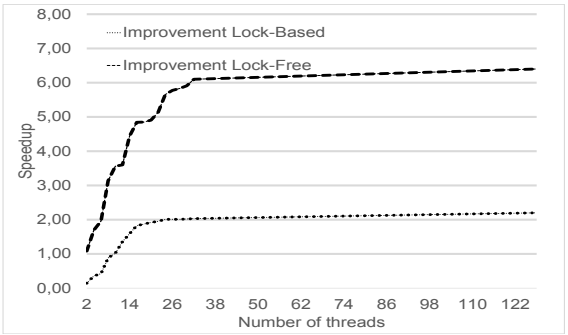

Figure 6: Speedup compared to our approach for write operations.

Surprisingly, the *lock-free* hash map is even slower than the traditional *lock-based* approach in writing operations. This is mainly because the *lock-free* method has to spinlock-wait until it can CAS the *key-value pair* between reading operations. The *lock-based* approach was implemented using boost mutexes, which do busy-waiting, while claiming the writer-lock. They allow other threads to continue with their operations, explaining these results. Figures 7 and 8 also illustrate the concrete timings, facilitating our findings above.

Our approach is almost independent of the number of concurrent threads. Consequently, the performance boost, compared to both other competitors, increases with an increasing number of threads (See Figures 9 and 10).

The only bottleneck of our method is the copy-on-write mechanism during write operations because we have to clone the current data. Figure 11 shows the dependency of the approach performance with respect to the *key-value* pair size. You can see a very slow decrease of the performance that would be only remarkable for very memory intensive pairs, i.e. pairs that require more than 10 kByte.

For retrieving key-value pairs (Figure 5) from the database, our approach is at least 10 times faster than the traditional blocking approach. As more threads access the database, the speedup increases even more.

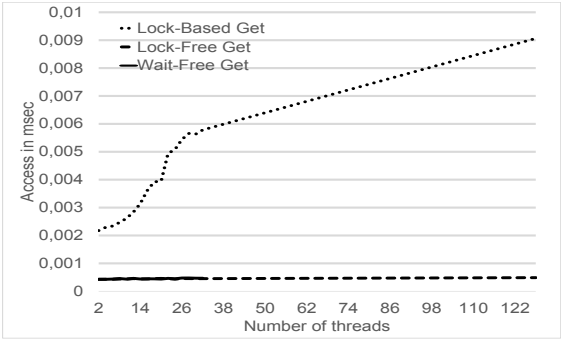

Figure 7: Timings for reading key-value pairs.

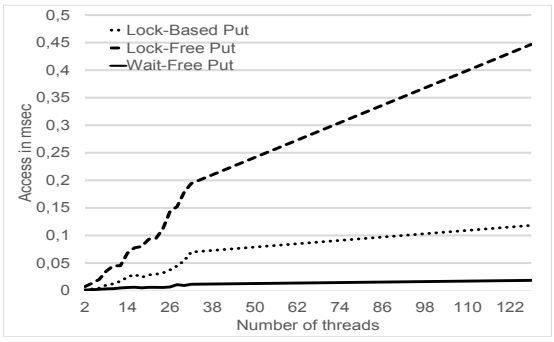

Figure 8: Timings for writing key-value pairs.

With regard to the lock-free hazard pointer implementation, we observe that no speedup can be gained, since both mechanisms offer wait-free read access. For storing key-value pairs (Figure 6) in the database, our framework also outperforms the other two approaches. It is 2 times faster (with respect to the lock-based approach), respectively 7 times faster (when compared to the lock-free implementation). We have to remark that the traditional lock-based approach is faster than our framework, when using less than 8 threads because the clone call needs more time as the unique lock reclamation. When using more than 8 threads, our approach again outperforms the lock-based one.

Figure 9 and 10 show how the concrete timings of the approaches change, when using 4 or 32 threads. The illustrations go along with our previous findings, indicating the performance decrease of the lock-based and lock-free approach. They explain the performance boost of our framework. Finally, Figure 11 shows that our approach incurs only a very small performance decrease while the key-value pair size increases. It decreases very slow but it would be remarkable when using pairs with high memory demand (e.g. more than 10 kByte), as the clone call of the copy-on-write mechanism mainly determines the access timing.

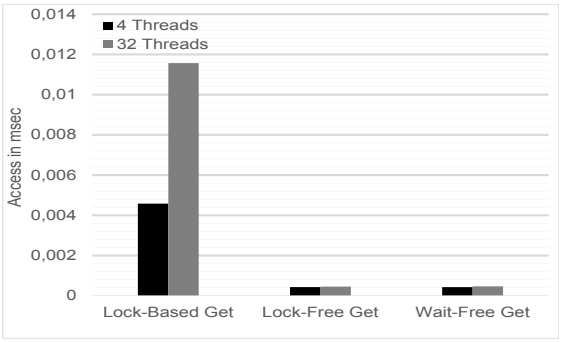

Figure 9: Timings compared to our approach for reading operations.

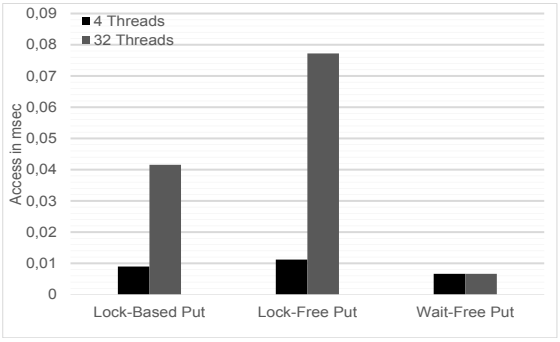

Figure 10: Timings compared to our approach for writing operations.

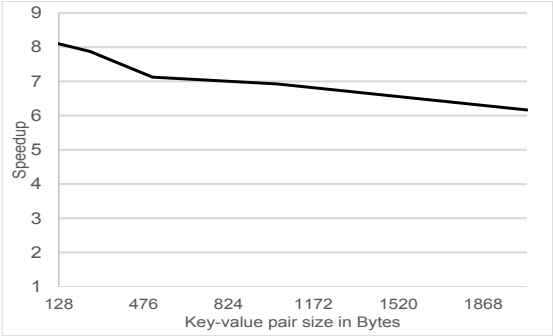

Figure 11: Dependency of the performance from the size of the data packages with respect to the lock-free implementation.

6 CONCLUSION

We presented a new approach that allows *wait-free* data exchange in massively threaded virtual environments. Our new method guarantees both, *wait-free* writing and reading access for concurrent threats. Moreover, we allow, unlike most other *wait-free* approaches, arbitrary data types to be written and read. Our method is easy to implement and the software architecture is highly maintainable. Adding new components that share new data with new and existing components is very simple and straight forward.

Moreover, we have implemented our approach and applied it to real-world VEs, but we also made synthetic benchmarks to compare it to other methods. Our case study proves that our framework is perfectly suited for real-time massively concurrent data exchange between arbitrarily simulation entities acting as concurrent software components inside an VR system with very low latency. Our comparison results, based on synthetic benchmarks, show that our new approach outperforms traditional *blocking* methods by more than an order of magnitude. It is still more than six times faster than the competing modern *lock-free* data structure.

We are confident that our technique can be applied easily to many other VR systems, including game engines. However, there are more avenues for future work. For instance, we will carry out further research on thread failures. In these cases it could happen that no memory could be released because the release function may not have been called. Moreover, our approach currently supports only a single producer for each key-value pair. In the future we hope to overcome this limitation. Another improvement could be a distinct hazard pointer list for each individual key-value pair, to facilitate key-based hazard pointer lists for faster hazard pointer management.

7 REFERENCES

[Fel13] Steven Feldmann, Pierre LaBorde, Damian Dechev. Concurrent Multi-level Arrays: Wait-free Extensible Hash Maps. International Conference on Embedded Computer Systems: Architectures, Modelling, and Simulation (SAMOS XIII), 2013.

[Her91] Maurice Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages and Systems, Volume 13, Issue 1, 1991.

[Her05] Maurice Herlihy, Victor Luchangco, Paul Martin, Mark Moir. Nonblocking memory management support for dynamic-sized data structures. ACM Transactions on Computer Systems, Volume 23, Issue 2, 2005.

[Har01] Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In Proceedings of the 15th International Conference on Distributed Computing, p. 300-314, 2001.

[Gid09] Anders Gidenstam, Marina Papatriantafilou, Hakan Sundell, Philippas Tsigas. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. IEEE Transactions on Parallel and Distributed Systems, Volume 20, Number 8, 2009.

[Det01] David L. Detlefs, Paul A. Martin, Mark Moir, Guy L. Steele. Lock-Free Reference Counting. In Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing, p. 190-199, 2001.

[Mic04] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Transactions on Parallel and Distributed Systems, Volume 16, Issue 5, 2004.

[Har07] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, Jonathan Walpole. Performance of memory reclamation for lockless synchronization. Journal of Parallel and Distributed Computing, Volume 67, Issue 12, 2007.

[Stel09] Philippe Stellwag, Alexander Ditter, Wolfgang Schröder-Preikschat. A Wait-Free Queue for Multiple Enqueuers and Multiple Dequeuers Using Local Preferences and Pragmatic Extensions. In Proceedings IEEE Symposium on Industrial Embedded Systems, p. 237-248, 2009.

[Tim12] Shahar Timnat, Anastasia Braginsky, Alex Kogan, Erez Petrank. Wait-free linked-lists. In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, p. 309-310, 2012.

[Neg12] Cristina de Negueruela, Michele Scagliola, Davide Giudici, Jose Moreno, Jorge Vicent, Adriano Camps, Hyuk Park, Pierre Flamant, Raffaella Franco. ARCHEO-E2E: A Reference Architecture for Earth Observation end-to-end Mission Performance Simulators. Simulation and EGSE facilities for Space Programmes, ESA ESTEC, 2012.

[Bra13] Anastasia Braginsky, Alex Kogan, Erez Petrank. Drop the Anchor: Lightweight Memory Management for Non-Blocking Data Structures. In Proceedings of the 25th Annual ACM symposium on Parallelism in algorithms and architectures, p. 33-42, 2013.