Scalable Concurrency Control for Massively Collaborative Virtual Environments

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

ABSTRACT

We present a novel concurrency control mechanism for collaborative massively parallel virtual environments that allows an arbitrary amount of components to exchange data with very little synchronisation overhead. The approach taken here is to maintain the shared world state of the complete virtual environment in a global key-value pool. Our novel method does not use any locking mechanism. Instead it allows wait-free data access for all concurrent components for both, reading and writing operations. This guarantees a highly responsive low-latency data access while keeping a consistent system state for all users and system components. Nevertheless, our approach is perfectly scalable even for massive multi-user scenarios. We provide a number of benchmarks in this paper, and the results show an almost constant run-time, independent of the number of concurrent users. Moreover, our approach outperforms previous concurrency control systems significantly by more than an order of magnitude.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming—*Parallel programming*

General Terms

Algorithms, Performance

Keywords

VR system architecture, wait-free, real-time, memory management, concurrency control

1. INTRODUCTION

Collaborative Virtual Environments (CVE) like massively multi-player or serious games, large-scale virtual cities or open space military training, have increased in popularity tremendously over the past years. All of these applications have in common that a large number of users interact *simultaneously* and in *real-time* in a *shared* virtual world.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
MMVE'15, March 18-20 2015, Portland, OR, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3354-2/15/03...$15.00
http://dx.doi.org/10.1145/2723695.2723699.

Interaction usually means that the user can manipulate objects in the virtual environment. In order to maintain a common and consistent state of the CVE for all users, interactions made by one user have to be made visible to all other users immediately. This requires a high *responsiveness* of the system, i.e. system changes have to be distributed with *low-latency*. Actually, experiments have shown that a bad responsiveness (high-latency) can lead to frustrated participant experience [5] and even to users completely losing interest in the application [1].

However, distributing shared data with low-latency is not enough to keep a shared virtual environment plausible and fair. A second challenge is the *consistency* of the system. This challenge is essential if several users are allowed to manipulate the same object simultaneously, e.g. in a serious multi-player game, when a group of users jointly solve tasks, or in collaborative virtual sculpting tools. Here, interactions of one user directly influence the simultaneous interactions of the other users. Finally, the demand for larger CVEs, i.e. virtual environments that allow a higher number of participants, more AI components, or more interactive objects, increased significantly during the past years. Consequently, modern CVEs should allow a high *scalability* in order to be prepared for today but also future applications. Obviously, these three key requirements of each CVE – *responsiveness*, *consistency* and *scalability* – are not independent of each other. For instance, *low-latency* is the prerequisite for the *consistency* while higher *scalability* is often contradictory to high *responsiveness*. Usually, this functionality of handling all simultaneous user interactions and allowing access to the common parts of the shared virtual environment is called *concurrency control management* (CCM). This CCM enables and maintains parallel, dynamic behaviour of the CVEs *shared world state*. A perfect CCM should fulfill all three partly conflicting requirements. Most current CCMs use a simple locking mechanism for simultaneous access to shared objects: if a user wants to manipulate an object, he locks the object, manipulates it, and when he has finished his manipulation, he releases it. This mechanism guarantees a consistent system and avoids race conditions; but if many users try to access the same object, it results in a bad responsiveness because other users have to wait until they gain access to the object, and it limits the scalability. Actually, locking mechanisms serialize concurrent user access, hence, they are only limited CCMs.

In this paper, we introduce a new concurrency control mechanism that fulfils all today's needs in massively shared data handling: our data structure is able to handle an arbitrary number of components accessing the shared world state, it achieves high responsiveness and a high scalability even for massively multi-user CVEs and it guarantees a fair and consistent access to shared objects in the virtual world.

In detail, our contributions are

- a novel *wait-free* data structure that guarantees *simultaneous read* operations without the need of locking the resources during read access
- a novel *wait-free* write mechanism that allows even *simultaneous write* operations, given some mild conditions that are often met in CVEs. For instance, it can be performed for all linear functions.

These contributions lead to a novel generic CCM for massively parallel CVEs with arbitrary applications which satisfies scalability, consistency and responsiveness.

The basic approach is a global key-value pool that maintains the world state of all shared objects. Our data structure is easy to implement and it supports insertion and removal of new resources — e.g. objects, users, AI components, etc. — during run-time while requiring only little memory. Our results show that our new CCM outperforms classic approaches significantly by more than an order of magnitude.

2. RELATED WORK

A distinctive characterization of CCMs is whether they are *locking* or *non-locking*. *Locking* approaches allocate resources exclusively by using various well-studied techniques such as mutexes, semaphores or condition variables. A main advantage of locking CCMs is that they avoid race conditions and naturally guarantee consistency of the system.

Most classic CCMs like [3] relied directly on standard *locking approaches*. Unfortunately, [17] reported that the locking approach scales only to at most ten peers on a local area network. This is mainly because of the problem that concurrent threads have to wait until a resource has been released. This may result in a loss of efficiency because problems like thread starvation or deadlocks can occur. Consequently, more modern CCMs like [8], [13] and [15] tried to avoid this problem by extending the basic locking mechanism. [15] used a simple first-come-first-serve locking, in which a central server granted manipulating access to a user on request. All other participants could only work on a local copy. But their local changes were not transmitted to the server site. Hence, only one user of the CVE could really interact whereas all other users could only observe the changes. [8] further presents a lock-based approach for the special case of collaborative sculpting. They split a mesh into different regions. For each region, a lock could be acquired. This allowed several users to work in parallel on the same mesh but at different parts. However, only one user could modify a region at the same time, multiple access was again not possible due to the lack of an suitable algorithm which could solve for parallel access. Additionally, the approach does not scale well with the number of users because the lock acquirement is slow. Filtering approaches basically offer a more general approach such as [10] [6]. The main idea is to reduce the acquirement latency by restricting the number of users which can request a lock. To do that, constraints have to be defined that are used to filter the requests. Even if the basic idea is generic, the constraints have to be adjusted for each individual application. Moreover, defining constraints which can be not met by all users simultaneously is challenging if not impossible in CVEs. Therefore, these filtering constraints are not generally valid for all kind of CVE applications. They also also do not solve the inherent problem of lock acquirement latency. Other approaches for collaboration in virtual environments or generic virtual reality system architectures (such as [2], [5], [7] or [16]) neglect the problem of efficient data access to an shared world state.

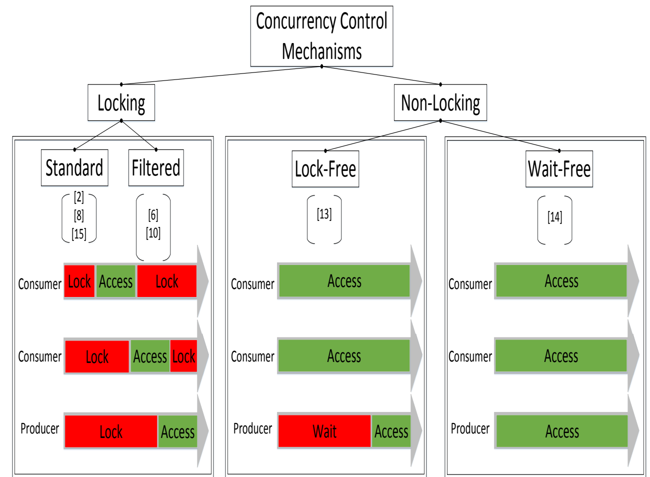

Figure 1: Classification of CCMs. Left: traditional locking approaches; threads can be arbitrarily delayed. Middle: lock-free approaches that only delay the producer. Right: our novel wait-free approach that does not delay other threads at all.

Non-locking approaches avoid this exclusive allocation of resources by introducing new data structures, mostly using a few atomic operations [4], [12], [11]. Non-locking approaches can be further classified into *lock-free* and *wait-free* methods. Both approaches avoid locks when solving concurrent access. Lock-free approaches guarantee progress of at least one of the threads accessing the shared data structure. All other threads can be arbitrarily delayed. In most approaches, all reading operations can happen in wait-free manner whereas all writers are delayed. Unfortunately, this can lead to thread starvation of the writers. Wait-free approaches guarantee access to the shared data structure in a finite number of steps for each thread, regardless of other threads accessing the shared data structure [9]. Unfortunately, most of them are restricted to a single writer [14]. In CVEs, simultaneous writing operations are essential, e.g. the calculation of users health points in games which are concurrently modified by several components of the system (such as enemy damage, healing or item buffs). Additionally, most wait-free approaches support only built-in primitive types and none of them was ever applied to CVEs, except [14].

Our new data structure presented here is based on this data structure. We extend it to support also multiple simultaneous wait-free write operations. Moreover, we present technical extensions that improve the performance significantly. We will start with a short recap.

3. RECAP: THE KV-POOL

The basis of our novel CCM is the wait-free approach presented in [14]. The core is a global dictionary, called key-value pool (*KVPool*), a centralized data storage that maintains the complete shared world state of the virtual environment.

Each component that wants to share data to other components in the CVE registers this shared data to the KVPool. Examples for such shared data are the life points of each player or AI in a game, meshes that can be modified by several users simultaneously, or transformations from simulation data. Registering shared data means the creation of a key-value pair (*KVPair*) in the global KVPool with a unique key which is required for fast identification. If components want access to the data, they simply have to pass this key to the KVPool. Each KVPair can be composed of arbitrary content, such as vectors, matrices, arbitrary numerics or mesh data. Consequently, one KVPair can have an arbitrary amount of member data which makes a KVPair universally usable. Each KVPair can either be accessed for writing or reading. We call components that modify the data, i.e. components that need writing access, the *producers* and those components that only read the produced data, *consumers*. In order to guarantee a wait-free reading access, we use a *copy-on-write* mechanism for data access: each KVPair maintains two copies of the data, a *producer reference* and a *consumer reference* (more precisely, the producer reference is allocated on the fly when a component requests writing access). If one or more consumers want to read data simultaneously, they simply pass the key to the KVPool and it returns a pointer to the consumer reference. Consequently, this guarantees wait-free reading access.

Producers get a pointer to the producer reference. When they finished their writing operations, they release the reference and it will become the new consumer reference, i.e. new consumer queries will directly routed to the updated data in the new consumer reference. However, this may result in race conditions if a consumer has not finished its reading of the old consumer reference (or if a KVPair is deleted completely).

Consequently, we introduced a guard mechanism to avoid this problem: the KVPool generates a *hazard pointer* that indicates an ongoing reading operation every time a consumer accesses a KVPair. When the consumer has finished its reading, the hazard pointer is released and the old consumer copy can be safely deleted. All hazard pointer are managed globally by the KVPool. These hazards are no locks, they are just pointers to a memory address of a consumer copy which should not be deleted.

The KVPool allows wait-free concurrent access. This results in a dramatic speed-up of several orders of magnitude compared to traditional lock-based approaches (see [14] for detailed timings), while avoiding all their problems like deadlocks or thread starvation. Moreover, it overcomes the many-to-many interface problem of standard VR system architecture approaches by introducing a centralized dictionary. Finally, it works completely asynchronous, no global main loop to synchronize concurrent access to the data is required.

However, the approach has two major drawbacks: first, it is restricted to a single producer for each KVPair. Hence, it can be hardly applied to simultaneous modifications of shared data that often appears in CVEs, like the aforementioned collaborative sculpting or game applications. Obviously, the approach can be easily combined with traditional locking-based mechanisms for parallel write operations for the same KVPair. However, this would partly neglect its advantages and inject their problems. Therefore, we propose a different solution. The second drawback is the global hazard pointer management. In the original implementation, we used a STL container to store the hazards. However the attachment and the deletion of hazards takes more time the more hazards have already been generated and stored. A filter-based approach to this hazard pointer management would help to optimize the approach.

4. OUR APPROACH

In this section, we present our new concurrency control mechanism that overcomes the limitations of the original KVPool. Namely, we present a novel method that avoids the slow global hazard pointer management, and we present a mechanism that also allows wait-free write access for several concurrent producers. Moreover, we give an overview on the implementation details.

4.1 Local Markers

In the original implementation of the KVPool, we used hazard pointers to avoid race conditions. If a consumer wants to read a consumer copy, it generates a hazard pointer that will be destroyed when the reading operation has been finished. All hazard pointers are stored in a global list of the KVPool. We used a wait-free list with atomic operations to organize the global hazard list. The main problem with this implementation are the relatively large amount of memory – each reading operation requires a pointer – and the time that is needed to release the pointers – we have to walk through the list to find the pointer before we can release it. Moreover, after each hazard release, we have to search the list if there are more hazards pointing to the same consumer copy in order to decide whether to delete the copy or not.

The main idea to avoid this problem is to use *local markers* instead of global hazard pointers. Basically, a local marker is a single integer value stored with each consumer copy. If a consumer wants to read the copy, it simply increments this local marker with an atomic operation. When it has finished the reading, it decrements the marker. Obviously, the local copy can be deleted if and only if the marker equals zero, because in this case, no reader is reading it any more.

The advantage is that we need only one single atomic integer for each consumer copy and not a hazard pointer for each consumer that access the copy. Moreover, the time consuming search for other hazards accessing the same copy can be omitted and we do not need complicated data structures like the wait-free list.

4.2 Wait-Free Concurrent Write Access

The original implementation allows only a single producer like most other wait-free data structures. In this section we present an extension of our KVPool also to multiple concurrent producers. The basic idea is to keep the copy-on-write mechanism but to allow several producer copies.

In detail, if a producer wants to modify data in a KVPair, the KVPool sends it a copy of the current data. Each concurrent producer gets its individual copy and all copies are labeled with a time code. Obviously, concurrent reading copies will get the same time code. After finishing the

modification, the producers informs the KVPool by writing back the data. Now, two possible cases may happen: first case, both the current consumer copy and the new producer copy have the same time code. In this case, we can simply replace the consumer copy and assign the current time code to it.

The second case is more interesting: the time code of the consumer copy and the producer copy are different. This means, the consumer copy has been already replaced by another producer. In this case, we *combine* the data of both copies to a new consumer copy. In order to combine both copies, the creator of the KVPair has to define an individual merge function which is used by the KVPool to solve the conflict.

This function allows a high flexibility and it covers almost all cases that usually happen in CVEs. For instance, it can simply overwrite all values, take the maximum or minimum of both values, in case of life points in computer games it can summarize points or in case of collaborative sculpting it can perform a linear interpolation between the modified vertices. Obviously, as a special case, it can implement the first-come-first-serve strategy of [15] by simply keeping the first change and throwing away all others, but also the local lock mechanism described by [8].

Several producers are allowed to write back their copies simultaneously. Because of this, we have to buffer this back-writing with a wait-free queue for each KVPair. Copies that are put into the queue are merged sequentially by the KVPool, outside the producers and consumers thread scope, using the merge function. Consequently, this sequential merging does neither influence the wait-free read, nor the wait-free write capability of our data structure. All components can still access all KVPairs for reading and writing.

4.3 Implementation Details

Figure 2 illustrates the above stated local marker concept as well as the relationship of producers and consumer with respect to the KVPairs stored in the KVPool. In the following, we will describe the KVPool and KVPair functionalities in more detail.

The KVPool, implemented as a hash map, offers only two access functions for the components: `put` and `get`. If consumer wants to read a value, it calls the `get` function and the KVPool returns the current consumer copy. Moreover, it increments the local marker (See Algorithm 1). If the consumer has finished reading, a `release` function will be called that decrements the local marker again. Additionally, it checks whether the local marker is zero and, probably, allows the deletion of the consumer copy if no other consumers still reads it.

Algorithm 1 KVPool::get(key,access)

```
 1: if key not in map then
 2:     return empty
 3: else
 4:     pair(KVDataPair) slot = map.getValue(key)
 5:     if access is producer then
 6:         return slot.producerreference.clone
 7:     else
 8:         slot.consumerreference.localMarkerIncrement
 9:         return slot.consumerreference
10:     end if
11: end if
```

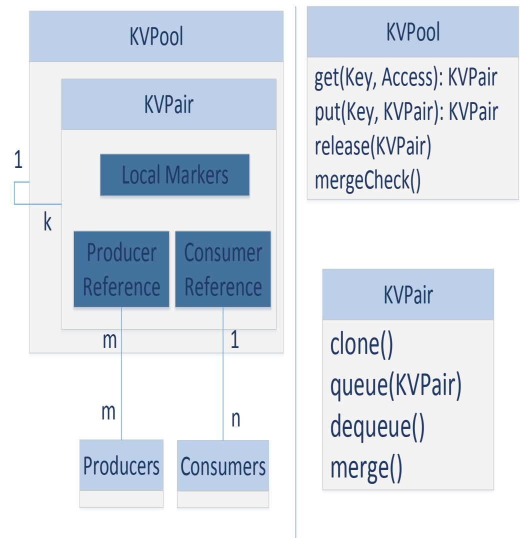

Figure 2: Relationships between key-value pool (KVPool), key-value pairs (KVPair), markers, producers and consumers.

Writing access also begins with a call of the `get` function. However, in this case, the KVPool returns a clone of the producer copy. When the producer has finished writing, it calls the `put` function (See Algorithm 2). The `put` function ensures the above stated wait-free reading access: it either replaces the old consumer copy by the new value or it collects those KVPairs that need to be merged as shown line 9. Actually, the compare of the time code has to be performed atomically in order to avoid race conditions during concurrent writes. Collected KVPairs are put in a queue that the KVPool maintains for each entry individually.

Additionally, the `put` function allows the insertion of new KVPairs to the KVPool. Moreover, it returns the old consumer reference as retired. This allows the deletion of this pair by the `release` function.

Algorithm 2 KVPool::put(key,value)

```
 1: KVPair retired
 2: if key in map then
 3:     pair(KVPair) slot = map.getValue(key)
 4:     if value.timecode = slot.producerreference.timecode
        then
 5:         slot.producerreference = value
 6:         KVPair retired = slot.consumerreference
 7:         slot.consumerreference = value.clone
 8:     else
 9:         slot.producerreference.queue(value)
10:         retired = slot.consumerreference
11:         pool.notify
12:     end if
13: else
14:     map.insert(pair(key,value))
15: end if
16: return retired
```

If the `put` function recognizes concurrent writes, i.e. a queue for a KVPair is not empty, the KVPool calls a `mergeCheck` function (See Algorithm 3) that processes the merges of those KVPairs.

Algorithm 3 KVPool::mergeCheck()

```
1: while slots are being marked do
2:    for all marked slots of map do
3:       slot.producerreference.dequeue
4:       slot.consumerreference = slot.producerreference.clone
5:    end for
6: end while
```

Each KVPair can store arbitrary data, hence, the merge function can have arbitrary composition. In the current implementation of our CCM, we provide six generic basic functions which can be used: min, max, average, addition, subtraction and replace. Each function operates on the defined member variables of the KVPair which should be merged. Obviously, the users can implemented their own functions to provide the required merge for their KVPair. As an example, Algorithm 4 shows an arithmetic mean merge.

Algorithm 4 merge-mean(KVPair a, KVPair b)

for annotated KVPair member variables mv of a and b **do**
  a.mv = $\frac{a.mv+b.mv}{2}$
end for

5. RESULTS

We have implemented our new CCM in C++. We performed 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.

The KVPool contained 50.000 KVPairs, each representing a virtual object. We performed 20,000 read- and write operations for each test. Each test was additionally repeated 50 times and we averaged the resulting timings. The access to the KVPool was modelled with different numbers of concurrent components, ranging from 4 to 512. We applied two test scenarios. In the first scenario the components were equally divided into producers and consumers of the KVPairs. The second scenario involved more producers, namely twice as much as consumers. To prevent caching, we inserted for each test run the KVPairs at random positions. The key size was set to 12 Bytes.

We compared the performance of our new approach to four different existing methods, which we adopted to the KVPool scheme. The first competitor was a standard locking scheme based on the boost locking library. The second approach was a typical filtered concurrency locking implementation based on [8]. The third competitor was an optimistic concurrency locking implementation which simply recomputes the values in case of a concurrent write based on [6]. Finally, we compared our approach to the original approach from [14].

Figures 3 and 4 show a comparison of the performance with respect to read and write access. In this test, producers read and write while consumers only read random KVPairs.

Our novel approach outperforms all other competitors. Obviously, the speed-up of our approach increases with an increasing number of components accessing the KVPool. Our approach is almost independent of the number of concurrent components. Additionally, our approach outperforms also the original approach from [14]. For less than approximately 28 components that concurrently access the KVPool, the filter-based approach performs almost identically to our approach. However, if more than 30 components access the KVPool, our approach, and even the original approach [14], easily outperform the filter-based approach.

Figure 5 shows a comparison of the memory usage with respect to a distributed KVPair with 128 Byte size. Surprisingly, our approach uses *less* memory than the original KVPool implementation, though using multiple copy-on-write clones for the concurrent write operations. Consequently, we expected a higher memory usage. However, our new improved KVPool performs much better due to the local markers. This results in a faster release of retired KVPairs and benefits therefore the overall memory demand of our new approach.

Figure 6 shows the performance gain of our approach with respect to the local marker concept by comparing the timings of hazard pointer acquisition and marker usage for the above stated test cases. In average, the local marker concept performs nearly twice as fast as the original hazard pointer approach. Overall, the local marking concept makes approximately 16% of the overall performance gain. The multiple wait-free writing constitutes 84% of the overall performance boost.

With respect to the overall performance, we can conclude that our approach outperforms the competitors (in average between a factor of 2 and 35) while using less memory than the original approach (on average 74%).

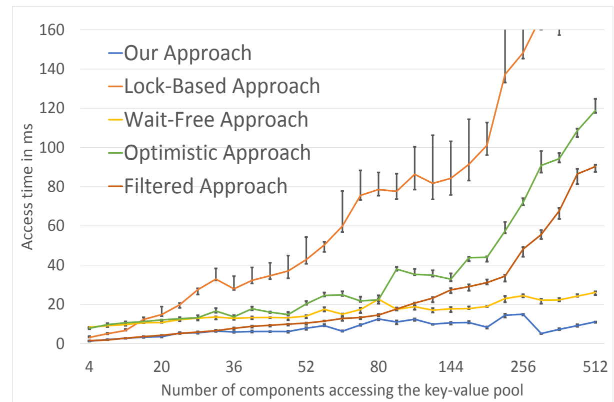

Figure 3: Timings for a combined read and write operation with an equal producer consumer distribution.

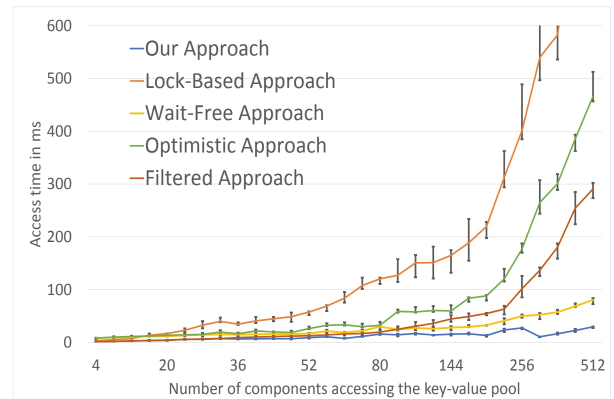

Figure 4: Timings for a combined read and write operation with twice as much producers as consumers.

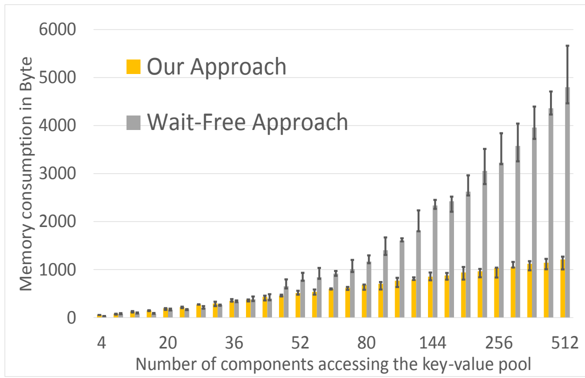

Figure 5: Memory consumption of our and original wait-free approach.

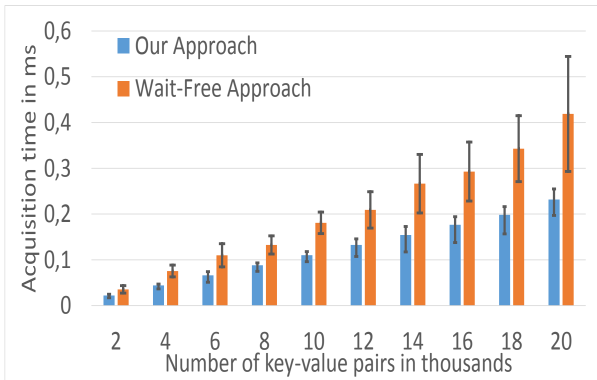

Figure 6: Performance gain of our approach with respect to the original wait-free approach.

6. CONCLUSIONS

We have presented a novel concurrency control mechanism for massively-parallel CVE applications. Our approach allows all concurrent threads complete wait-free access for reading and writing operations and it requires no locks like most other approaches reported in the literature. Our new CCM is easy to implement and supports arbitrarily complex data types and the memory overhead is small. Adding or removing new concurrent components is very simple and straight forward, even during run-time.

Our benchmark shows that our approach outperforms other CCMs known from the literature significantly, especially in massive parallel scenarios with a large number of concurrent components.

However, our synthetic benchmark was restricted to a single PC, but we are confident that our approach scales well for distributed applications like massive multi-player games or other distributed CVEs. In a distributed CVE application, the KVPool would serve as a central time synchronising host, which updates the timestamps of the KVPairs. Typical distributed CVE problems such as network delays would not affect read or write operations. Delayed write operations would be merged and delayed read operations would still retrieve the newest data from the KVPool. However, a fast network interface of the central host would be needed to avoid possible bottlenecks when introducing such a centralized server solution. Additionally, it would be desirable to define a theoretic basis for generic merge functions. Until now, we just used functions that we needed in our applications, like min, max or linear combinations. Defining a general valid merge for arbitrary data would be more challenging, if general possible. In addition to that, a concurrent quality measure for data could define the boundaries and constraints for arbitrary data for the merge. Finally, it would be desirable to remove also the last remaining "sequential" step on processor execution level, the atomic operations. Probably, this could be done with unique prime identifiers for the components. These identifiers could be used for unique data access determination instead of using atomic markers. This could happen in parallel, in separation of the actual KVPair read and write operation.

7. ACKNOWLEDGMENTS

This research is based upon work funded by the German BMWi under grant 50NA1318.

8. REFERENCES

[1] R. A. A. Boukerche, N. J. McGraw. A grid-filtered region-based approach to support synchronization in large-scale distributed interactive virtual environments. *International Conference on Parallel Processing Workshops*, pages 525–530, 2005.

[2] C. V. L. Arthur Valadares, Thomas Debeauvais. Evolution of scalability with synchronized state in virtual environments. *International Workshop on Haptic Audio Visual Environments and Games*, pages 142–147, 2012.

[3] O. H. Christer Carlsson. Dive - a multi-user virtual reality system. *Virtual Reality Annual International Symposium*, pages 394–400, 1993.

[4] G. L. S. David L. Detlefs, Paul A. Martin. Lock-free reference counting. *ACM Symposium on Principles of Distributed Computing*, pages 190–199, 2001.

[5] R. W. David Roberts. Controlling consistency within collaborative virtual environments. *International Symposium on Distributed Simulation and Real-Time Applications*, pages 46–52, 2004.

[6] S. H. Dongman Lee, Mingyu Lim. Atlas - a scalable network framework for distributed virtual environments. *Presence*, 16:125–156, 2007.

[7] K. H. Frank Steinicke, Timo Ropinski. A generic virtual reality software system's architecture and application. *ICAT Proceedings International Conference on Augmented Tele-Existence*, pages 220–227, 2005.

[8] F. F. N. Frederick W.B. Li, Rynson W.H. Lau. Vsculpt: A distributed virtual sculpting environment for collaborative design. *IEEE Transaction on Multimedia*, 5:570–580, 2003.

[9] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, pages 206–209, 1991.

[10] D. L. Jeonghwa Yang. Scalable prediction based concurrency control for distributed virtual environments. *Virtual Reality*, pages 151–158, 2000.

[11] M. M. Maurice Herlihy, Victor Luchangco. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems*, 23, 2005.

[12] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 16, 2004.

[13] S. O. Olarn Wongwirat. Performance evaluation of compromised synchronization control mechanism for distributed virtual environment. *Virtual Reality*, 9:1–16, 2006.

[14] G. Z. Patrick Lange, Rene Weller. A framework for wait-free data exchange in massively threaded vr systems. *Journal of WSCG 2014*, 22:383–390, 2014.

[15] B. H. Pietro Buttolo, Roberto Oboe. Architectures for shared haptic virtual environments. *Computers & Graphics: Haptic Displays in Virtual Environments and Computer Graphics in Korea*, 21:421–429, 1997.

[16] M. K. Tom Feldmann. Vair: System architecture of a generic virtual reality engine. *Computational Intelligence for Modelling, Control and Automation - International Conference on Intelligent Agents, Web Technologies and Internet Commerce*, 2:501–506, 2005.

[17] K.-Y. W. Un-Jae Sung, Jae-Heon Yang. Concurrency control in ciao. *IEEE Proceedings Virtual Reality*, pages 22–28, 1999.