Wait-Free Hash Maps in the Entity-Component-System Pattern for Realtime Interactive Systems

Patrick Lange* Rene Weller† Gabriel Zachmann‡

University of Bremen

ABSTRACT

In the past, the Entity-Component-System (ECS) pattern has become a major design pattern used in modern architectures for Realtime Interactive Systems (RIS). In this paper we introduce high performance wait-free hash maps for the System access of Components within the ECS pattern. This allows non-locking read and write operations, leading to a highly responsive low-latency data access while maintaining a consistent data state. Furthermore, we present centralized as well as decentralized approaches for reducing the memory demand of these memory-intensive wait-free hash maps for diverse RIS applications. Our approaches gain their efficiency by Component-wise queues which use atomic markup operations for fast memory deletion. We have implemented our new method in a current RIS and the results show that our approach is able to efficiently reduce the memory usage of wait-free hash maps very effectively by more than a factor of ten while still maintaining their high performance. Furthermore, we derive best practices from our numerical results for different use cases of wait-free hash map memory management in diverse RIS applications.

Index Terms: H.5.1 [Information Interfaces and Presentation]: Multimedia Information System—Artificial, augmented and virtual realities D.4.2 [Operating Systems]: Storage Management—Garbage collection

1 INTRODUCTION

A central part of Realtime Interactive Systems (RIS), Virtual Reality (VR) systems, game engines and realtime simulations is the generation, management and distribution of the global simulation or world state. Usually many independent software components need to communicate and exchange data in these modern graphics interactive systems in order to generate this global state [17]. These components and their corresponding performance within an RIS is often governed by the functional as well as non-functional requirements of typical RIS development such as (realtime) performance, responsiveness, scalability, consistency and (re-)usability [19]. Consequently, RIS development strives for reusable patterns and software architectures in order to increase the satisfaction of the above mentioned requirements, especially for massive amounts of RIS software components [12, 14]. Hence, diverse approaches have been presented to tackle this kind of challenge. In the past, the Entity-Component-System (ECS) pattern has become a major design pattern used in modern architectures for RIS [7]. This pattern strives for high reusability and architectural scalability. The main idea of ECS is to decouple high-level modules such as physics, rendering or sound from the low-level objects with their corresponding data. Therefore, ECS introduces three software architectural objects: *Entities*, *Components* and *Systems*. These are used to describe objects of a RIS via composition instead of object-oriented inheritance.

Many advantages arise when using ECS. For instance, sthe behavior of *Entities* can be changed at runtime by adding or removing *Components*. Moreover, *Systems* are likewise interchangeable as they are not part of the *Entity* nor *Component* implementation. This allows even the quick swap of e.g. a physics engine. This leads to the elimination of ambiguity problems of wide and deep inheritance hierarchies, often encountered in traditional RIS development. As a result of this, the ECS pattern and variations of it have been applied to many RIS, such as [16, 18, 12].

However, the ECS pattern does not aim at satisfying the performance requirement of RIS architectures as it does not specify any low-level implementation. For instance, the *System* access implementation to the *Components* is not defined. Usually, every *System* iterates over a container (e.g. a array) of all *Entities* and applies the *Systems* behavior on the corresponding *Components*. In fact, modern RIS can consist of hundreds or thousands of *Components* and *Systems*. Therefore, an access parallelization (e.g. in the form of threads or OpenMP support) is necessary in order to maintain realtime performance of the whole application. Consequently, this container of *Components* (and therefore every *Component*) has to be implemented as a shared data structure. Such shared data structures can quickly become bottlenecks of modern RIS applications as they typically use crucial software synchronisation patterns such as mutexes, semaphores and consequently synchronization which can lead to thread starvation or system deadlock [17]. The ECS pattern does not cover any guidelines or specifications for effectively solving this problem but several applicable concurrency control management approaches exist in the literature.

One promising approach are wait-free concurrency control managements. They guarantee access to a shared data structure in a finite number of steps for each *System* (eg. as a traditional thread or OpenMP implementation), regardless of other *Systems* accessing the shared data structure. This means that these approaches do not need any traditional locking mechanism in order to preserve a consistent data state. Experiments have shown a superior performance of wait-free approaches compared to traditional locking approaches (See Figure 1). Consequently, wait-free approaches deliver high performance access even for massive numbers of concurrent read and write operations. However, as a drawback they often have a large memory footprint [17, 19]: Usually, they rely on a double-buffering approach with atomic operations in order to achieve wait-free behavior. This double-buffering creates for every write access to the shared data structure a clone of the manipulated data. When all read operations on the cloned data are finished, the cloned data is released.

*e-mail:lange@cs.uni-bremen.de

†e-mail:weller@cs.uni-bremen.de

‡e-mail:zach@cs.uni-bremen.de

Hence, the amount of cloned data directly responds to the amount of write operations [17, 19]. In this paper, we present a novel solutions to this challenge; our new approaches allow concurrent read- and write access even for highly data driven RIS applications (resp. RIS applications which inherit many *Components* and/or *Systems*). Moreover, they can even handle multimodal RIS applications in which different *Systems* interact with each other in different frequencies.

In detail, our contribution of this paper is

- an extension of the ECS pattern for high performance double-buffered wait-free hash maps with
- centralized as well as decentralized approaches for efficient memory management of these data structures which greatly reduces their memory consumption.

Our contribution allows non-locking read and write operations of *Systems*, leading to a highly responsive low-latency data access while maintaining a consistent state even for structured *Components*. Simultaneously, our contribution greatly reduces the memory footprint of the introduced wait-free hash amps. Our novel memory management is easy to implement and it fits perfectly into the implementation of wait-free hash maps without altering the ECS pattern itself.

Our approach therefore greatly benefits the overall RIS performance.

2 Related Work

Research in increasing scalability and performance as well as managing concurrency within RIS frameworks has attracted increasing interest in the last decade. This research can be broadly classified into two classes: high-level and low-level concepts. High-level concepts describe the overall RIS software architecture in terms of software classes which use standard libraries for solving parallelization and concurrency of the low-level implementation. Examples for such high-level concepts are Simulator X [12] with its actor model [22] or other high-level approaches, e.g. based on dataflows [23]. Further, [2] gave an overview of high-level architectures aiming at solving consistency and concurrency for Collaborative Virtual Environments (CVE). However, this overview neglected wait-free synchronization approaches. In contrast to these high-level concepts, low-level concepts investigate programming language specific implementations of synchronization approaches for RIS challenges. In the past, low-level concepts mainly introduced lock-based concurrency control management (CCM) approaches for RIS, VR and CVE frameworks in order to efficiently implement the high-level concepts.

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.

Many traditional RIS, especially CVEs, such as [3, 24] used lock-based approaches until [24] reported that the locking approach scales only to at most ten components. 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 CVEs like [9, 15, 21] tried to avoid this problem by extending the basic locking mechanism, e.g. by a first-come-first-serve locking [21]. Further, more sophisticated concurrency control approaches introduced fine-grained locks per object for single-write and multiple-read operations [10, 8, 5]. Due to the limitations of lock-based approaches, wait-free approaches based on hash maps for RIS had been introduced [17, 19, 20].

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 by introducing a few atomic operations [13]. This means that these approaches do not need any traditional locking mechanism in order to preserve a consistent data state. Experiments have shown a superior performance of wait-free approaches with respect to traditional locking approaches as Figure 1 illustrates. These wait-free approaches not only support structured *Components* such as arrays or lists but also use fast hash key operations in order to find and retrieve the stored *Component* inside the used hash table. Due to their excellent scalability, they are perfectly suited for RIS frameworks which need to support massive amounts of *Systems*, such as multi-agent system based VR and RIS applications [25, 18]. Consequently, using wait-free data structures as a data access backbone can highly improve the performance and scalability of RIS frameworks.

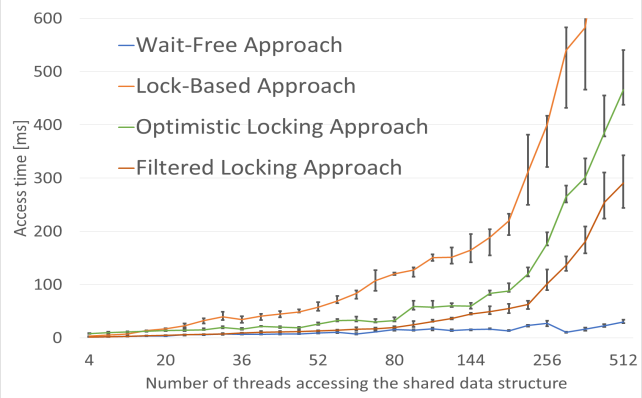

Figure 1: Performance comparison of wait-free and lock-based concurrency control management implementations, which are traditionally used for VR and RIS frameworks. Adopted from [19].

However, these wait-free hash maps come at a cost: In order to achieve wait-free behavior of read and write operations, they use double-buffering for write operations. This means that every write access on the shared data structure is preceded by a double-buffering which clones the data [17, 19]. All ongoing read operations can still access the old data state while new read queries are directly routed to the new (manipulated) data state. Therefore, after a given timespan, the old data will not be used any more. When all read operations on the old data state are finished, the data is released. Hence, the amount of cloned data directly responds to the amount of write operations [17, 19].

In the following, we will describe the integration of double-buffered wait-free hash maps into the ECS pattern. Furthermore, we will describe our novel memory management for these data structures which reduces their memory demand greatly. Therefore, our integration and novel memory management overcomes the limitations of the presented related work.

3 Wait-Free Hash Maps for the Entity-Component-System Pattern

In this section we will give a short recap of the ECS pattern and introduce our actual implementation that relies on double-buffered wait-free hash maps.

3.1 The Entity-Component-System Pattern

In the past, the Entity-Component-System (ECS) pattern has become a major design pattern used in modern architectures for RIS [7]. The main idea of ECS is to decouple high-level modules such as physics, rendering or sound from the low-level objects with their corresponding data. Therefore, ECS introduces three software architectural objects: *Entities*, *Components* and *Systems*:

- The *Entity* is a general purpose object which is usually defined as a unique id. These *Entities* can be further described via composition of *Components*.
- The *Component* is the raw data for one aspect (e.g. a position, velocity or sprite) of general purpose objects.
- The *System* performs global actions on every Entity that possesses a *Component* with the same aspect as that *System*. Each *System* thereby runs continuously (e.g. as a thread).

These objects are used to describe objects of a RIS via composition instead of object-oriented inheritance. The traditional way to implement simulation or game objects within RIS was to use object-oriented programming. Each object was modelled and implemented within a typical class hierarchy which intuitively allowed for an instantiation of these classes. This enabled simulation or game objects to extend to other objects through polymorphism. However, with an increasing complexity of the RIS, this leads to large, rigid class hierarchies. These wide and deep hierarchies become consequently increasingly difficult to maintain. In addition, placing a new simulation or game object into the hierarchy is further complicated if the object needs a lot of different types of functionality from different domains. Figure 2 illustrates this limitation in a game-based RIS scenario. Usually, the conflicting code is then moved to the base class which results in super classes. These super classes gradually decrease the maintainability and scalability of the overall RIS architecture.

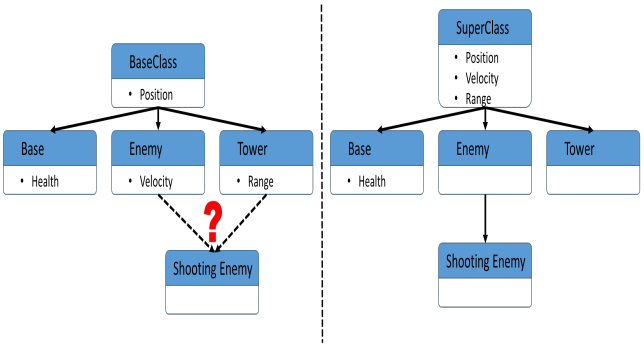

Figure 2: Evolving problems in inheritance based object design in RIS applications: In order to preserve class-wise consistency, super classes are constructed.

Typically, these deep and wide inheritance structures can be vertically decomposed with the ECS pattern (see Figure 3). This allows greater flexibility and adaptability in defining simulation or game objects (e.g. vehicles, sensors, enemies, etc.) as every object is an *Entity*. Every *Entity* consists of one or more *Components* which add aspects (e.g. position, velocity, sprite, etc.) to the *Entity*. Within this context, the behavior of an *Entity* can be changed at runtime by removing or adding *Components*.

Furthermore, even *Systems* are decoupled as each *System* applies its computation on the same *Component* types which are referenced via *Entities*. As an example, think of a physically-based simulation for gravitational forces: The corresponding *System* will apply Newton's law of gravity everytime on the same *Component* types: the position and velocity but it does not concern any more properties of the *Entity*.

As a consequence, even *Systems* can be easily added, removed or changed as they are not part of *Entity* or *Component* implementation. Actually, object-oriented problems of deep and wide inheritance structures as mentioned above are eliminated.

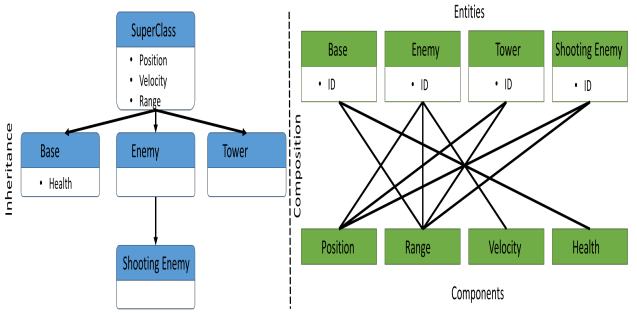

Figure 3: The Entity-Component-System pattern: Deep and wide inheritance structures arise in traditional RIS development (left). The ECS pattern decouples the data and algorithms via composition (right).

3.2 Integration of Wait-Free Hash Maps

As previously stated, the ECS pattern does not specify the low-level implementation of the *System* access to the *Components*. In this paper, we investigate the applicability of wait-free hash maps for this *System* access to the *Components* as they promise high performance even for massive numbers of concurrently acting *Systems*.

Several wait-free hash maps for RIS related research have been proposed such as traditional hash maps [17, 19] or graph-based hash maps [20]. These wait-free hash maps can be easily integrated in to the ECS pattern as follows: All *Components* (which can represent primitive or structure data) are stored inside the wait-free hash map. These *Components* are accessible via a unique key which is generated for every *Component*. All *Systems* and *Entities* can refer to these *Components* via their unique keys which retrieve the *Component* from the hash map. Every *Entity* has a list of keys which defines the *Component*-wise composition. Adding and removing *Components* from the *Entity* are implemented as insertion and deletion operations on this key list. Every *System* iterates over the *Entities* and uses the stored keys in order to retrieve the corresponding *Components* for computation. This integration of wait-free hash maps does not alter the original ECS approach. Figure 4 further illustrates this concept.

Overall, the *System* access to all *Components* is implemented straightforward (e.g. with OpenMP support) (See Algorithm 1). Note, that no locking operations are needed in order to maintain consistency of the hash map and consequently of all *Components*.

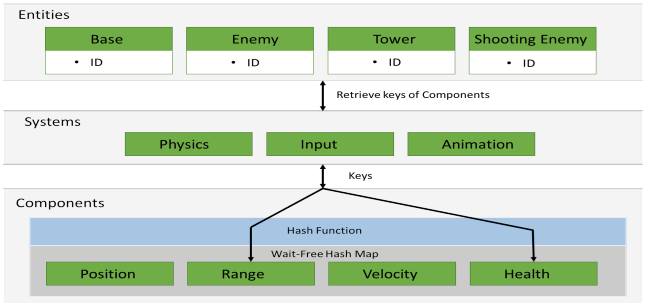

Figure 4: Integration of (wait-free) hash maps into the ECS pattern: Components reside inside a hash map which is concurrently shared by all Systems.

Listing 1: System access on Entities and Components in C++ pseudocode

```
// Define OpenMP parallelization with x threads
#pragma omp parallel for num_threads(x)
for (int i = 0; i < Entities.size(); i++)
{
  if (Entities[i].Type == System.Type)
  {
    ComponentKeys = Entities[i].WriteKeys
    for (int j = 0; j < ComponentKeys.size(); j++)
    {
      Component c = Hashmap.get(ComponentKeys[j])
      System.ApplyComputation(c)
      Component clone = Hashmap.set(c)
      // Delete clone after all concurrent
      // read operations have finished
    }
    ComponentKeys = Entities[i].ReadKeys
    for (int l = 0; l < ComponentKeys.size(); l++)
    {
      Component c = Hashmap.get(ComponentKeys[l])
      c.incrementReadMarker
      // Use data as long as needed
      // without altering it
      // ...
      c.decrementReadMarker
    }
  }
}
```

The *Component* access of the double-buffering wait-free hash map is implemented in accordance to [17, 19]: Every System can retrieve a *Component* from the hash map by looking up the corresponding key. Within the double-buffering approach, every *Component* is stored as a producer and consumer version in the hash map. For all read operations, the hash map returns a pointer to a dedicated consumer version of the *Component*. Consequently, all read operations work on the same memory as they can not affect each other. All actively reading *Systems* notify their access by incrementing (read operation starts) and decrementing (read operation has ended) an atomic marker of the consumer version. For write operations, *Systems* retrieve a producer version of the *Component* which is a different memory object than the consumer version. After modifying a *Component*, a *System* can notify its changes to all other *Systems* by storing the *Component* back to the hash map. This write process uses the aforementioned double-buffering and returns the cloned *Component*.

In detail, the complete write operation of a *System* can be decomposed into six steps: A *System* wants to modify a *Component* which is stored inside the hash map. To do that, the corresponding hash map access returns the producer version of the *Component*. The *System* can then modify the *Component* and stores it back to the hash map. In order to notify all other *Systems* about these modifications, this write operation creates automatically a clone of the producer version which is used as the new consumer version. All concurrent read operations are routed to the old consumer version as long as the actual write operation of the hash map lasts. When the new consumer version is available, all concurrent read operations are routed directly to the new consumer version. In the meantime, the old consumer version is not deleted to prevent memory failures. When all ongoing read operations on the old consumer data are finished, the corresponding memory will be deleted, hence completing the double-buffering principle. Parallel write operations are merged as described in [19]. Figure 5 illustrates this double-buffering approach. The main challenge remains the efficient release of the old *Component* data.

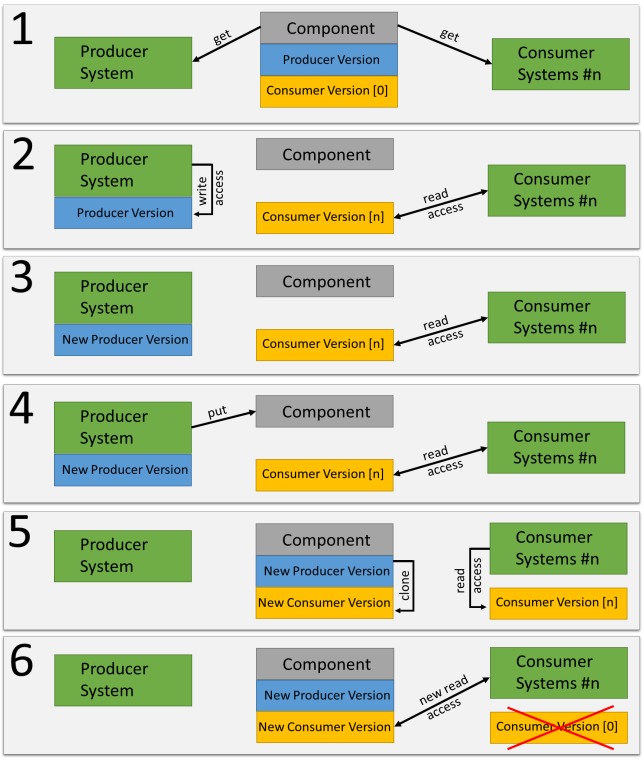

Figure 5: The double-buffering approach applied to the ECS pattern, simplified into six steps: Every write access is preceded by a cloning process of the *Component* data. When all parallel read operations are finished, the old data is deleted.

4 Memory Management for Double-Buffered Wait-Free Hash Maps

In this section we present centralized as well as decentralized approaches for the memory management of cloned *Component* data in wait-free hash maps for the ECS pattern. First, we present two centralized approaches based on periodic memory release as well as threaded memory release. These approaches are located in the central hash map itself and rely on finished read notifications that are triggered by all *Systems*. Second, we present a decentralized approach which defines an individual memory release per *System*. In this case, every *System* itself takes care of releasing unused *Components*.

All approaches follow the presented main principle of atomic memory markup [17, 19]: All *Systems* notify each other when they read or write data via notifications. This notification is implemented as an atomic marker which is increased when the read access begins and which is decreased when the read access has finished. Consequently, if this atomic marker is zero, the data can be safely deleted.

The basic challenge is the management, i.e. the saving and the efficient release, of the generated *Component* clones. The original implementation [17] introduced a single list implementation within the centralized hash map. This list was used to store every cloned data and was periodically checked. However, the nature of RIS applications leads to different generation, update and deletion frequencies of different *Components*. For instance, a collision detection query is updated at 1000 Hz and an animation of a scene node is played when a certain event has triggered in the RIS. In these cases the resulting *Components* (e.g. the resulting collision volume and rotation matrix) are more frequently or rarely updated. Consequently, different *Components* types (e.g. player-input, physically-based simulation results or animators) are more frequently updated

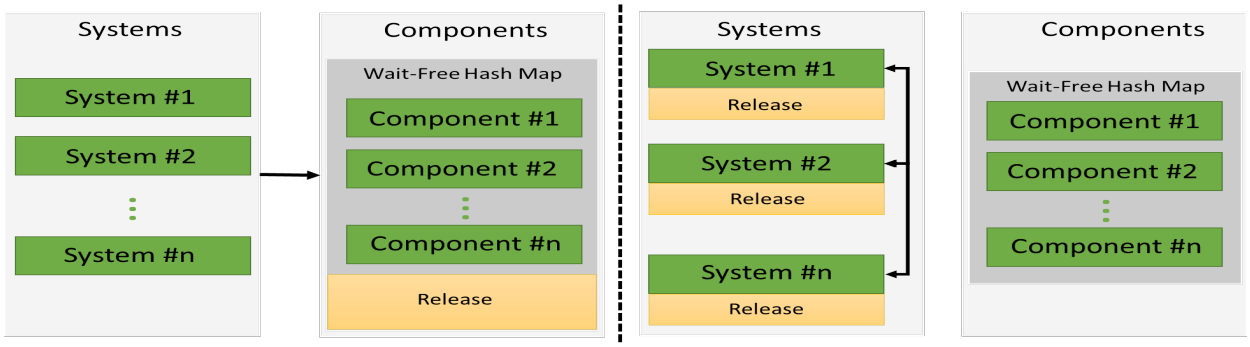

Figure 6: The centralized and decentralized memory management approaches: In the centralized approach, every *System* notifies the central hash map via atomic markers. The hash map itself releases the memory in either periodic or threaded implementation (left). In the decentralized approach, all *Systems* notify themselves about ongoing and finished read operations. In addition, every *System* itself takes care of releasing its cloned Component data (right).

than other *Components* types. This means that also the corresponding clone data is more frequently generated. In this case, it is desirable to handle the memory release per *Component* type. This leads to our approach which introduces *Component*-wise queues for storing the cloned *Component* data per Component type. These queues basically split all cloned *Components* into smaller chunks which can be faster checked than a single container for all cloned *Components*. Every cloned *Component* data is directly sorted into the corresponding queue immediately after its creation. Additionally, every queue itself has a atomic boolean markup. Every time a *System* notifies a finished read operation (by decrementing the corresponding atomic marker of the *Component*), this queue markup will be set to true. The marker will be set to false when the release function has finished its checks (See Algorithm 2). This release function iterates over all queues and checks their boolean markers. If a queue marker is set to true (resp. a *System* has finished its reading operation on a cloned *Component* data within the queue) it will further iterate the *Component* queue. After a queue check, the release function will set the corresponding queue markup to false. Figure 7 illustrates this queue concept. Algorithm 2 shows how these *Component*-wise queues can be iterated in order to efficiently release the unused cloned *Component* data.

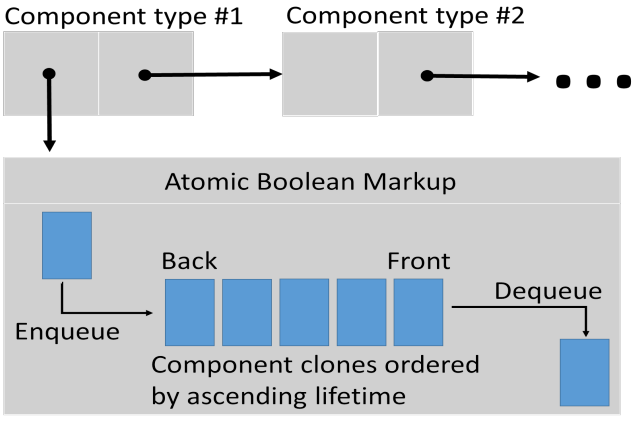

Figure 7: The *Component*-wise queue concept: Cloned *Component* data is stored into their corresponding queues which enable fast memory deletion.

Listing 2: Cloned Component data access via queues in C++ pseudocode

```
for (int i = 0;
     i < ComponentQueues.size();
     i++)
{
  if (ComponentQueues[i].Markup == true)
  {
    for (Component c = ComponentQueues[i].peek();
         ComponentQueues[i].size() > 0;
         c = ComponentQueues[i].peek())
    {
      if (c.ReadMarker == 0)
      {
        ComponentQueues[i].pop()
        delete c
      }
    }
    ComponentQueues[i].Markup = false
  }
}
```

The iteration over these *Component*-wise queues in order to minimize the amount of useless memory checks remains challenging. As mentioned above, we present two approaches for tackling this challenge: centralized and decentralized.

The centralized memory management approach is implemented in three variations: periodic, continuously threaded and on-demand threaded. In all cases, the memory management is located within the central wait-free hash map and all *Systems* notify the hash map with respect to the above mentioned atomic memory markup. The continuously threaded approach implements a complete separate thread that runs constantly in parallel within the RIS and checks the queues as shown above. The on-demand threaded approach implements a complete separate thread that is only activated when one *System* has finished its operations and generated new cloned data. After one check for all queues, it goes back to sleep state until it is notified by another *System* again. The periodic centralized approach is called in accordance to the *System* frequencies. *Systems* usually apply their computations to the *Components* periodically: The physically-based simulation (e.g. collision detection) runs traditionally at 1000 Hz while animations require 30 Hz or 60 Hz. Hence, it is also favourable to directly couple the frequency of the memory release with the actual implemented *System* frequencies of the RIS. We propose three frequencies for the periodic memory release: First, the memory release can be performed at the slowest frequency of all *Systems*.

Second, the memory release can be performed at the fastest frequency of all *Systems*. At last, the memory release can be performed at the average frequency of all *Systems*. The periodic approach builds upon application domain knowledge. For instance, if it is known to the RIS developer that mostly fast-paced physically-based simulations are present in the application, also a rapid periodic check for memory release could be useful and vice versa. In contrast, the decentralized memory management is located within the *System* implementation. All *Systems* notify each other with respect to the above mentioned atomic memory markup. Every *System* actively checks on his own after each completed computation whether memory can be deleted or not. Figure 6 illustrates these approaches.

5 Use Case Study

Our approach enables the implementation of very different categories of RIS applications. Exemplarily, we present the application to a high fidelity dynamics and spacecraft EDL (entry, descent and landing) end-to-end spaceflight mission simulator [16, 1].

More precisely, we implemented a simplified version of ESAs ARCHEO-E2E system [4] 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 within the ECS architecture. Further, all *Components* are stored inside a wait-free hash map and the *Systems'* access to the *Components* is implemented as presented above. Several *Entities* are present in the RIS as the sensor input (e.g. camera and range finder measurements) 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 *Components*. Further, the internal spacecraft components such as sensors, guidance, navigation and control read and write *Components* periodically. In addition to this, also the physically-based simulation as well as rendering act as a *System* in this application. Summarizing, this scenario has a large amount of concurrent read- and write operations of many interacting *Systems*. Figure 8 shows a rendering of the simulation in which a spacecraft conducts scientific experiments while orbiting an asteroid.

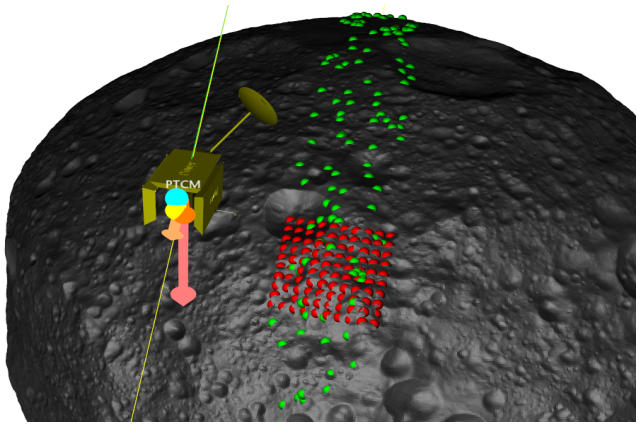

Figure 8: Use case study: A spacecraft is orbiting an asteroid. Rangefinder (red) and landmark (green) measurements are generated for spacecraft self-localization [11, 6] purposes.

6 Evaluation

The use case described above is the basis of our evaluation. We have implemented our memory management approach in C++. We performed our experiments on a machine with Intel Core i7 quad-core processor with Hyperthreading enabled and 8GB of memory.

We compared our different memory management strategies in several set-ups of our use case. Our evaluation concerned different amounts of active *Systems* within the use case study as well as varying *Component* types. The *Components* represented simple three-dimensional positions, 3x4 and 4x4 matrices, point clouds (ranging between 1,000 and 40,000 points), three-dimensional line segments and geometry as well as standard programming language objects such as strings or integers. Furthermore, the *Components* varied in size between a few Byte and several Megabyte. We performed 10,000 read- and write operations for each test. In order to avoid caching effects we repeated all tests 50 times and we averaged the resulting timings.

Figure 9 illustrates the performance of our novel memory management. Here, we evaluated how many unused *Components* are deleted each simulation step in the RIS. Clearly, our novel memory management outperforms the original implementation. Actually, all of our proposed strategies outperform the original implementation but they also perform diverse for varying numbers of *Systems*. Each implementation, whether centralized or decentralized, exhibits a sweet spot in which it outperforms the competitors. In detail, the decentralized approach outperforms the centralized implementations the more *Systems* access the *Components*. In case of less active *Systems*, the centralized approaches, especially the frequency dependent variations, perform better. We believe that an increasing number of *Systems* increases the overall memory dependency between the *Systems*. This means that the more active *Systems* a RIS inherits, the more *Systems* are likely to use the same *Components*. Therefore, they are "blocking" the release of the cloned *Component* data. Consequently, the responsibility of releasing the memory shifts from the overall collective of *Systems* more to the single *System*, resp. to the decentralized approach. At last, the centralized approaches always outperform the original implementation [17].

Furthermore, Figure 10 illustrates the lifetime of cloned data before it is deleted. Clearly, our novel memory management approach outperforms the original implementation [17]. In detail, the decentralized approach outperforms all competitors while the centralized approach with high frequency can nearly compete with it. It can be further observed that the frequency of the periodic centralized implementation directly relates to the lifetime of the unused *Component* data.

Finally, Figure 11 compares the actual access performance of the original wait-free hash map implementation with our implementation with enhanced memory management. Furthermore, we compared our wait-free implementation with a traditional lock-based implementation for the *Component* access. It can be observed, that our memory management does not introduce any performance issues. Both wait-free implementations behave almost identical for wait-free read and write operations. Furthermore, our results show that the wait-free access implementation gradually outperforms the traditional lock-based implementation with an increasing amount of *Systems* by several orders of magnitude.

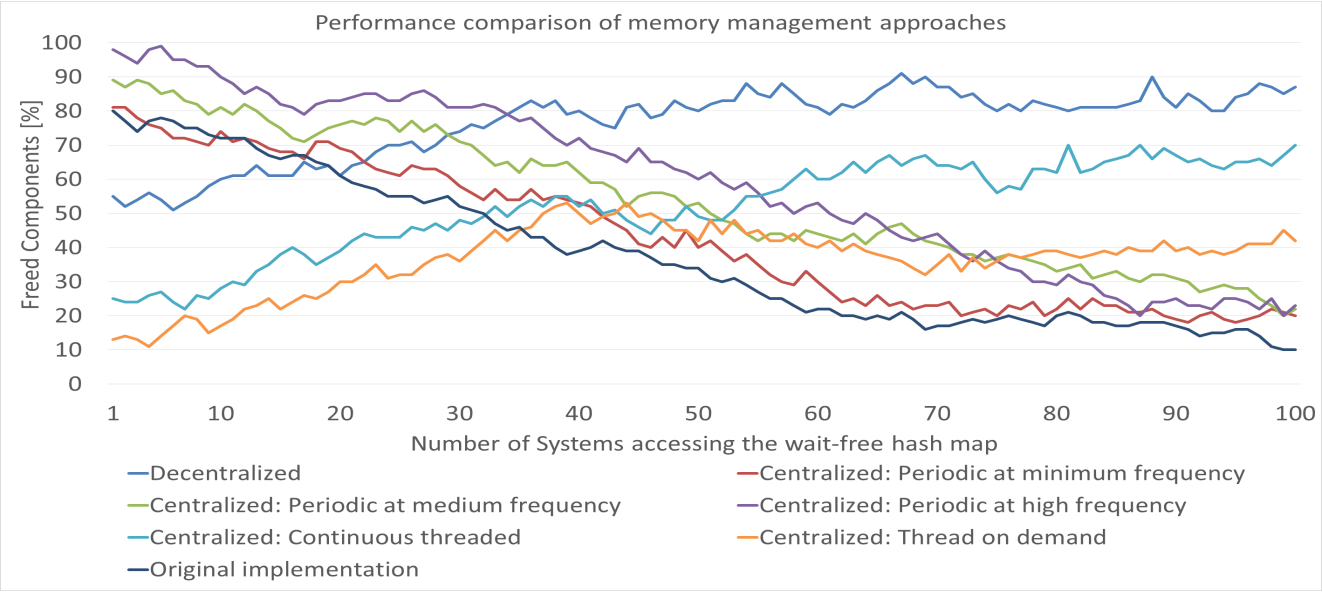

Figure 9: Performance comparison of the proposed memory management approaches.

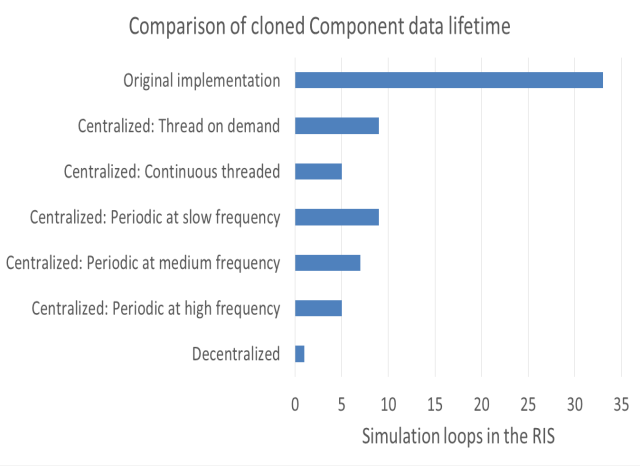

Figure 10: Comparison of the unused *Component* data lifetime before they are deleted in RIS setting with 100 *Systems*.

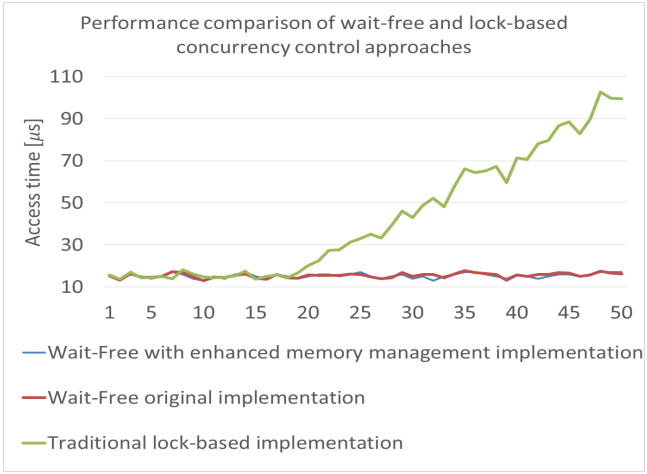

Figure 11: Performance comparison of our memory management enhanced wait-free implementation, original wait-free implementation and standard locking approach.

7 Best Practices

It can be summarized, that our evaluation revealed different advantages of the the presented memory management approaches. In detail, for only a few active *Systems* in the RIS, the centralized (periodic with any frequency) approaches outperform their competitors. However, if the RIS inherits more than 30 active *Systems*, the decentralized approach outperforms all competitors. Furthermore, the memory is deleted fastest in the decentralized approach, followed by the centralized (periodic with high frequency and continuous threaded implementation) approaches with an increasing number of active *Systems*. We can derive from this some best practices for RIS development which use wait-free hash maps with double buffering. Tables 1 and 2 illustrate our findings for different use cases, namely: RIS applications which inherit few/many *Systems* as well as RIS applications which inherit small (e.g. only primitives like vectors, matrices) or big *Component* data (such as large structured data such as point clouds or triangles).

Few *Systems*	
Small *Component* Data	Big *Component* Data
Centralized (periodic with any frequency) management	Centralized (periodic with high frequency) management

Table 1: Guideline for memory management approaches for few *Systems* within a RIS.

Many *Systems*	
Small *Component* Data	Big *Component* Data
Decentralized management	Decentralized management

Table 2: Guideline for memory management approaches for many *Systems* within a RIS.

8 Conclusion

We have presented a novel efficient memory management for high performance wait-free hash maps within the ECS pattern. Our approaches enable non-locking read and write operations in highly data driven RIS applications with large numbers of concurrent consumers as well as producers while simultaneously maintaining a consistent state of the whole system. Our approaches are easy to implement and the do not alter the ECS pattern itself.

Our results show that such wait-free implementations outperform traditional lock-based implementations by several orders of magnitude. Moreover, our new memory management based on double-buffered wait-free hash maps is able to reduce the memory consumption by a factor of ten without affecting the performance. All results were measured in a realistic high fidelity space robotics environment where we were able to achieve realtime performance even in complex situations. Hence, believe that our approach is applicable to a wide variety of RIS applications which use the ECS pattern or variations of it.

In the future, we would like to extend the double-buffered wait-free hash maps with intelligent *Component* cloning. This would incorporate that the write operation only clones the *Component* data when needed and not by default. This could be implemented with a *System*-based heuristic which notifies when the next read operation may happen. Furthermore, we would like to apply wait-free hash maps in more RIS applications in order to enable a profound analysis of their capabilities. Finally, we would also like to extend our approach with high-level concepts for adaptive memory management of double-buffered wait-free hash maps. It would be beneficial if a RIS framework itself could determine which memory management suits the current *System* and *Component* composition best. This would incorporate that the RIS framework monitors the usage of all *Components* from every *System* and adapts the memory management accordingly.

Acknowledgements

This research is based upon the project KaNaRiA, supported by German Aerospace Center (DLR) with funds of the German Federal Ministry of Economics and Technology (BMWi) under grant 50NA1318.

References

[1] Alena Probst, Graciela Gonzales Peytavi, David Nakath, Anne Schattel, Carsten Rachuy, Patrick Lange, et al. Kanaria: Identifying the Challenges for Cognitive Autonomous Navigation and Guidance for Missions to Small Planetary Bodies. *International Astronautical Congress (IAC)*, 2015.

[2] Cedric Fleury, Thierry Duval, Valerie Gouranton, Bruno Arnaldi. Architectures and Mechanisms to Maintain efficiently Consistency in Collaborative Virtual Environments. *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, 2010.

[3] Christer Carlsson, Olof Hagsand. DIVE - A Multi-User Virtual Reality System. *Virtual Reality Annual International Symposium*, pages 394–400, 1993.

[4] 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.

[5] David L. Detlefs, Paul A. Martin, Guy L. Steele. Lock-Free Reference Counting. *ACM Symposium on Principles of Distributed Computing*, pages 190–199, 2001.

[6] David Nakath, Carsten Rachuy, Joachim Clemens, Kerstin Schill. Optimal rotation sequences for active perception. In *Proc. SPIE Multisensor, Multisource Information Fusion: Architectures, Algorithms, and Applications 2016*. SPIE Press, 2016.

[7] Dennis Wiebusch, Marc Erich Latoschik. Decoupling the Entity-Component-System Pattern using Semantic Traits for Reusable Realtime Interactive Systems. *IEEE VR Workshop on Software Engineering and Architectures for Realtime Interactive Systems*, 2015.

[8] Dongman Lee, Mingyu Lim, Seunghyun Han. ATLAS - A Scalable Network Framework for Distributed Virtual Environments. *Presence*, 16:125–156, 2007.

[9] Frederick Li, Rynson Lau, Frederick Ng. VSculpt: A Distributed Virtual Sculpting Environment for Collaborative Design. *IEEE Transaction on Multimedia*, 5:570–580, 2003.

[10] Jeonghwa Yang, Dongman Lee. Scalable Prediction Based Concurrency Control for Distributed Virtual Environments. *Virtual Reality*, pages 151–158, 2000.

[11] Joachim Clemens, Thomas Reineking, Tobias Kluth. An evidential approach to SLAM, path planning, and active exploration. *International Journal of Approximate Reasoning*, 2016.

[12] Marc Erich Latoschik, Henrik Tramberend. Simulator X: A Scalable and Concurrent Software Platform for Intelligent Realtime Interactive Systems. *Proceedings of the IEEE VR*, 2011.

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

[14] Nicholas F. Polys, Sham S. Visamsetty, Puranjov Bhattacharjee, Eli Tilevich. The Value of Patterns in Deep Media Scenegraphs. *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, 2009.

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

[16] Patrick Lange, Alena Probst, Abhishek Srinivas et al. Virtual Reality for Simulating Autonomous Deep-Space Navigation and Mining. *24th International Conference on Artificial Reality and Telexistence (ICAT-EGVE 2014)*, 2014.

[17] Patrick Lange, Rene Weller, Gabriel Zachmann. A Framework for Wait-Free Data Exchange in Massively Threaded VR Systems. 2014.

[18] Patrick Lange, Rene Weller, Gabriel Zachmann. Multi Agent System Optimization in Virtual Vehicle Testbeds. *EAI SIMUtools*, 2015.

[19] Patrick Lange, Rene Weller, Gabriel Zachmann. Scalable Concurrency Control for Massively Collaborative Virtual Environments. *ACM Multimedia Systems, Massively Multiuser Virtual Environments (MMVE)*, 2015.

[20] Patrick Lange, Rene Weller, Gabriel Zachmann. GraphPool: A High Performance Data Management for 3D Simulations. *ACM SIGSIM Conference on Principles of Advanved Discrete Simulations (PADS)*, 2016.

[21] Pietro Buttolo, Roberto Oboe, Blake Hannaford. Architectures For Shared Haptic Virtual Environments. *Computers & Graphics: Haptic Displays in Virtual Environments and Computer Graphics in Korea*, 21:421–429, 1997.

[22] Stephan Rehfeld, Henrik Tramberend, Marc Erik Latoschik. An actor-based distribution model for Realtime Interactive Systems. *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, 2013.

[23] Thomas Knott, Benjamin Weyers, Bernd Hentschel, Torsten Kuhlen. Data-flow Oriented Software Framework for the Development of Haptic-enabled Physics Simulations. *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, 2014.

[24] Un-Jae Sung, Jae-Heon Yang, Kwang-Yun Wohne. Concurrency Control in CIAO. *IEEE Proceedings Virtual Reality*, pages 22–28, 1999.

[25] Yue Yu, Abdelkader El Kamel, Guanghong Gong. Multi-Agent based Architecture for Virtual Reality Intelligent Simulation System of Vehicles. 2013.