Should STL containers be used in game engines?

Jiri Holba

1 Introduction

Whether you want to start working on your own engine or you hop on a moving train and start using an existing one, some questions remain the same. Would you be better off with custom containers? When should you reach for a vector-like container instead of a map? Is a linked list the best choice when you need to insert and remove elements often? Is emplace faster than insert? And many more.

We are going to explore the options that the C++ language and its standard template library has to offer. We will look at a set and an unordered set as well as a map and an unordered_map. While performance is not the only thing to consider, it is going to be our main focus here. We will also take a look at what role the CPU cache plays in different scenarios, what cost we are paying when we decide to add a new element into a container, and when our instinctive choice of container with the search time complexity O(1) is actually worse than O(*n*).

We found that modern CPUs favor cache friendly containers. A vector or custom containers storing elements in contiguous memory should be our default choice unless we know for sure that we need to store a large amount of data.

2 CPU cache

The CPU cache is a small, very fast memory which is usually on the same chip as the CPU. It makes memory access times as short as possible (Hruska 17).

How much smaller and faster than main memory? Let us take a look at the current generation of game consoles. Both the PS4 and Xbox One use an AMD Jaguar architecture chip with two modules with 4 cores each. Each core uses a 32-Kbyte, 2-way set associative L1 instruction cache and a 32-Kbyte 8-way set associative L1 data cache. Each module has a 2-Mbyte 16-way set associative L2 cache shared by the 4 cores in that module. That gives the CPU 4 Mbytes of L2 cache in total while the memory is two thousand times larger at 8GB. The approximate times for load-to-use latency are 1ns for L1 cache, 15ns for L2 cache, and 50 – 100ns for main memory (IGN 16a, IGN 16b, AMD 13).

3 Performance measurement methodology

The tests we ran focus on STL containers used in game AI and gameplay. All tests were performed on containers with sizes from 1 to 100 elements because that is the number of elements usually used in gameplay and AI systems.

We ran tests on a system using an AMD Jaguar CPU described above. The CPU clock was set to 1.6 GHz and the system was using 8 GB of DDR3 memory. We measured

time intervals by reading the Time Stamp Counter register (TSC). The TSC offers the precision needed for the short intervals measured during these tests. It lacks precision for measuring longer time intervals such as hours or days, but that is not our case. TSC counts the number of cycles since reset, therefore the results are not in nanoseconds but in cycles. Results in cycles can give the reader also a better idea of how many instructions the CPU could have processed while waiting for the memory.

Every single test was repeated 10,000 times in two different variants:

1. "Warm cache" – during these tests, all memory allocations needed for container construction were done without any allocation between them and the test was performed right after the container was constructed and filled with testing data. By doing so, we increased the chance that the data needed for the test was in L1 cache or at least in L2 cache and it was not scattered around memory. That corresponds to a situation in a game where we create the whole container at once and we access it multiple times in a tight loop after that.
2. "Cold cache" – these tests were closer to a real-life example. In real life, the cache gets trashed by other processes, and elements are not close to each other in the memory. Other allocations occur between allocating the two adjacent elements or the memory fragmentation forces the element's physical locations to be more scattered. We simulated these conditions by allocating random memory after each element insertion and by trashing the cache before each test iteration.

4 Set and unordered set

Set and unordered set are associative containers holding unique keys where the key is also an item value. When we try to insert an element which already is in the container, the insert request is ignored.

Set stores elements in a specific order given by the ordering criterion which is a property of the container. The default comparison function is std::less. Since elements are ordered, we can iterate on a subset of data limited by a value range. We can use a bidirectional iterator for going through all the elements in both directions. Set is typically implemented as a balanced binary search tree structure. The tree itself might have some extra properties to achieve a better performance during some operations. For example, it might store the number of elements to avoid traversing the whole tree during the "size" function call. These properties add some memory size overhead. All the following memory requirements are for a 64bit application using the MS Visual Studio 14.0 implementation of the standard library. The set object requires 16 bytes of memory; the pointer to the head node and size of the container. Every element added incurs an additional 26 bytes of overhead; a pointer to the right branch, a pointer to the left branch, a pointer to a parent, and two chars (one to mark the head of tree and one for color). VS14.0 implements the set as a red-black tree (Cormen 90). That gives us the size of the set with n elements expressed by Equation 1.

$$size = 16B + n * (sizeof(element\ type) + 26B + padding) \quad (1)$$

If our stored data type size is 4B or less, the whole element will take 32B, but if it is 8B or more, there is 6B of padding added and the element size goes up to 40B.

On the other hand, std::unordered_set uses a hash table for fast element access and it does not store data in any specific order. Elements are stored in buckets which are indexed directly by the hash value of the key. When we create an empty std::unordered_set object, it allocates a table for 8 buckets by default, meaning that it will contain 16 pointers since it has a "begin" and "end" pointer for each bucket list. It also creates an empty list for the elements. The empty list contains one invalid element. Its size is two pointers plus the size of our stored data type. In total, the memory requirement for an empty unordered set is 192B plus the size of the invalid element in the empty bucket which adds to at least 216B in total. Equation 2 shows the formula for the memory required for an unordered set for n < 8. The default size of the hash table is 8 buckets and when we add the 8th element, the table grows to 64 buckets by default.

$$size = 192B + (n+1) * (sizeof(element\ type) + 16B). \quad (2)$$

Besides a set and an unordered set, we have another option how to hold unique elements in a container. We can keep them in a regular std::vector and do a linear search to see if a new element is already present before adding it. The only memory overhead this approach has are the 24B needed for the member variables of the std::vector class. The memory requirements are shown in Formula 3.

$$size = 24B + n * sizeof(element\ type) \quad (3)$$

When it comes to the performance of operations, we know that the set is implemented as a balanced binary tree which gives us the time complexity O(log *n*). The unordered set implemented as hashing table has the average complexity of O(1), but it can have the complexity of O(*n*) in the worst case (i.e. when all elements land in the same bucket). Elements stored in the std::vector are searched linearly which gives us the complexity of O(*n*). The time complexity gives us an idea about how the time required for accessing an element is going to go up with the number of elements in the container. However, it tells us nothing about the actual time needed. Let us assume that we have one element only. In case of an unordered set, we need to calculate the hash for the key, use it as an index into the hashing table, load the pointer to the bucket stored in the hashing table, load the pointer to the first element in the bucket, load the first element and get its value. That is quite a lot of operations compared to what happens when we want to access the element in a vector where we need to get the pointer to data and load the first element. So in theory, the simple vector should outperform the std::unordered_set for size of 1. Let us confirm this theory by actually measuring it.

4.1 Average search

When we compare the results of the cold cache test shown in Figure 1 with results of the warm cache test in Figure 2, we clearly see that the cache plays an important role here. The

measured function calls are the same for both tests and they are shown in Listing 1. Please note that the values on the time axis are 10 times bigger in Figure 1. Each cache miss costs us a few hundred CPU cycles which could be used elsewhere.

Listing 1 Function calls measured in an average search test.

```
auto it = std::find(vector.begin(), vector.end(), key);
auto it = set.find(key);
auto it = unordered_set.find(key);
```

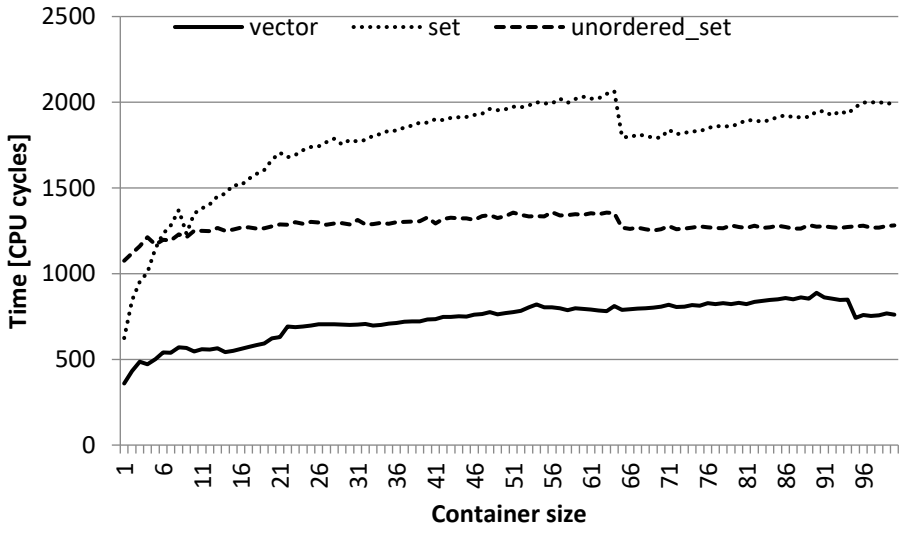

Figure 1 Vector beats other containers in average search time in the cold cache test.

When we take a look at Figure 2, which shows the results for cases when we tried hard to have everything in cache, the vector is still better for a smaller number of elements. The graph shows the average search time, but even if we measure the worst search time, it would be better for up to ~30 elements because it is easier for modern CPU to optimize.

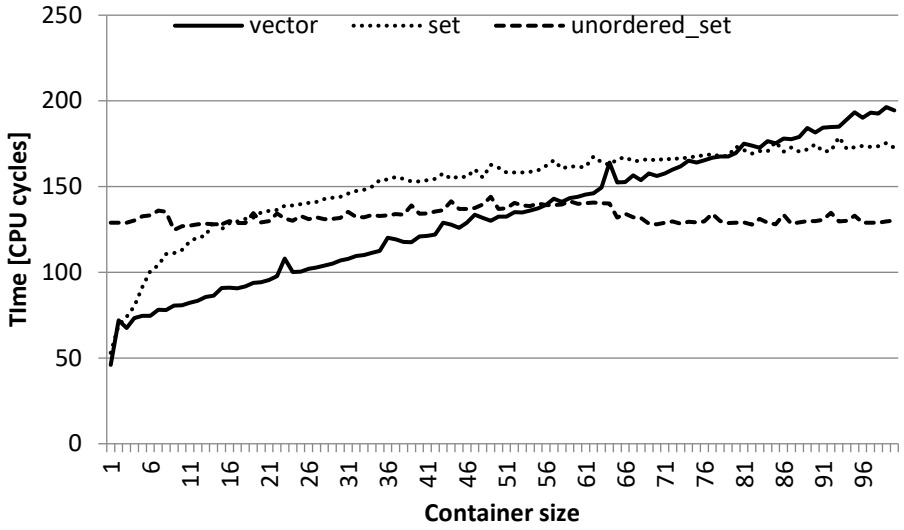

Figure 2 Average search times with warm cache nicely follow the time complexity function.

We have talked about the CPU cache only so far, but the optimizations don't end there. The CPU has many more mechanisms to process code faster. It can also have a branch predictor and out of order instruction execution support. Both mechanisms work better when going through a contiguous block of memory than, for example, traversing a tree where we need to check if we go right or left. In the tree traversal case, the prediction will fail more often than during the simple check if we need to continue in the linear search or not. There are two branches in the linear search code and both are easily predictable. One checks if we have reached the end of the container and the other one checks if we have found the correct element. Both checks are likely to be false. When the number of elements goes up the more checks we have to do and the time complexity will take its toll. With 60 elements to crunch through, the linear search has to iterate over 30 elements on average and check every one of them and at this point, the constant time complexity of the unordered set starts winning.

4.2 Insert

As we said earlier, the elements are unique and when we try to insert something that already is in the container, nothing is added and std::pair<bool, iterator> is returned. The bool value is false and the iterator points to the existing element. With that in mind, we can expect the insert time for element duplicates to be pretty much the same as the average search time. If we compare the insert time in Figure 3 to the search time in Figure 1 for the cold cache test, we can see that the insert time is a bit higher than the search time. The same thing can be observed when we compare the Figure 4 to the Figure 2 showing the warm cache results. That is because the algorithms are different. The search is looking for an element, while the insert is looking for a place where to insert a new item.

Listing 2 Function calls measured in insert duplicate element test.

```
if(std::find(vector.begin(), vector.end(),key) ==
vector.end()) {
     vector.push_back(key); }
set.insert(key);
unordered_set.insert(key);
```

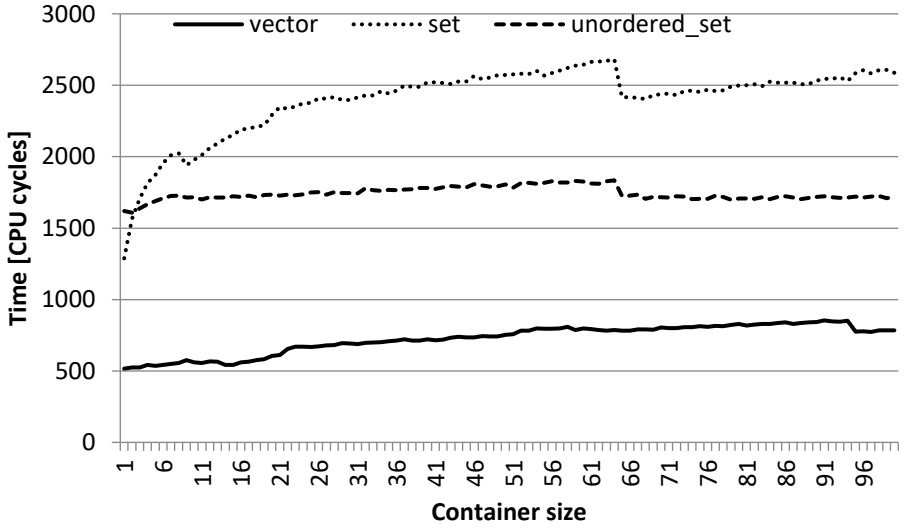

Figure 3 Insertion time of a duplicate object is more expensive than search if the cache is cold.

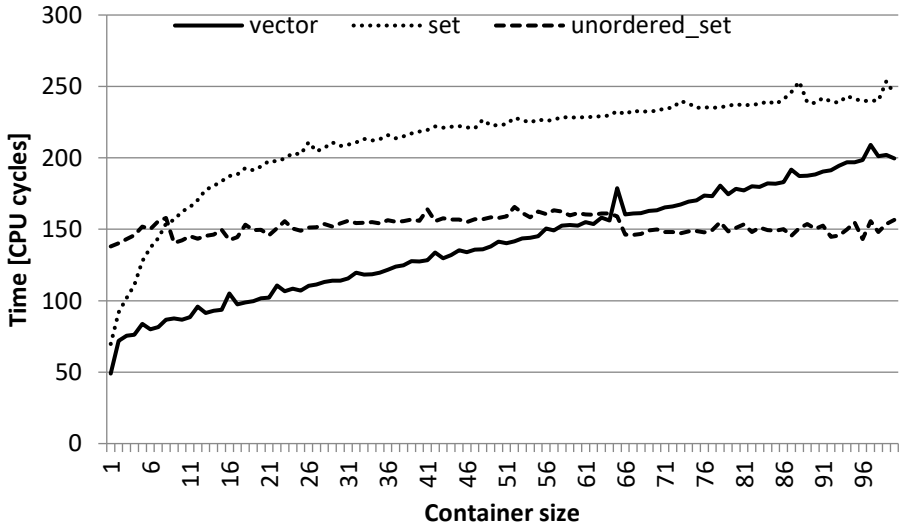

Figure 4 A vector has better insertion time for duplicate objects than a set even for bigger container sizes.

4.3 Emplace

C++11 gave us a very neat, new feature – move semantics. The idea is that when we want to move the ownership of an object, we don't have to create a copy and destroy the original. Instead, we move the content of the object from one place to another. Let us say that we have a class with a list as one of the constructor parameters. We can create two versions of the constructor. One which takes a const reference to the list and another one that takes an rvalue reference to the list. When the reference to the list is passed down into the constructor, it has to create a copy of the list. This means it will copy every single element in that list and add it into a new list. When we use the move semantic to pass down an rvalue reference though, the list can copy the original head pointer into the new instance and then set the original head pointer to nullptr. So in the end the move operation is a more efficient way to transfer the resources from one object to another. When we try to move a built-in type, the value is simply copied.

All the STL containers have a new set of functions to support the move operation. They support move construction, move assignment and have a few new functions for adding new elements as well. The insert function has an overload for rvalue parameters so it can use the move semantic and transfer the resources if possible. There is also a new function called emplace which also takes an rvalue parameter but with one important difference. It takes a constructor parameter for the value type held by the container and it creates a new element in-place instead of moving it or copying it there.

When we take a look at Figure 5 and Figure 6 we see surprising results for the performance of the emplace call. The function calls we measured are shown in Listing 3. The vector performed far better than the other two containers even when we measured it with a warm cache. The reason for that is that both a set and an unordered set have to create a new element before they figure out if it already is there or not. As we know, the element is also a key and therefore emplace has to create it first. A set as well as an unordered set needs to allocate a memory space for each element, so what we see on the graph is the price of memory allocation. Memory allocation is clearly very expensive, especially when the cache is cold. We get a cache miss for the lookup in our container and plenty of additional cache misses for the allocator itself. The same would happen in the emplace function of the std::map and std::unordered map, but C++17 provides a better interface for those by adding a new function try_emplace which takes a key as a first parameter and checks if the key is not already in the container before moving the value.

If we use a vector instead of a set or an unordered set, we can first check to see if the key is already present and insert a new element only when it is not. In our test scenario, we knew that the only parameter provided was the key itself, so we didn't need to create it. In case we want to wrap the vector and provide a more generic interface shared by our code base, we would probably need to create the object with the vector as well. But even so, we could create the object as the last one in our vector and remove it if it is present already. We would avoid the memory allocation unless the vector needs to grow.

Listing 3 Function calls measured in emplace duplicate element test.

```
if(std::find(vector.begin(), vector.end(),key) ==
vector.end())
     vector.emplace_back(std::move(key));
set.emplace(std::move(key));
unordered_set.emplace(std::move(key));
```

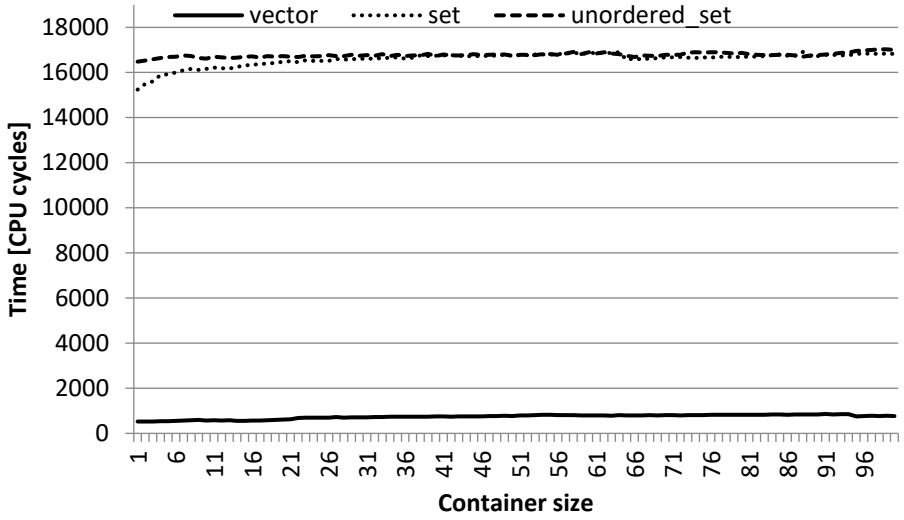

Figure 5 Emplace of the duplicate element is more expensive for a set and an unordered set in this cold cache test.

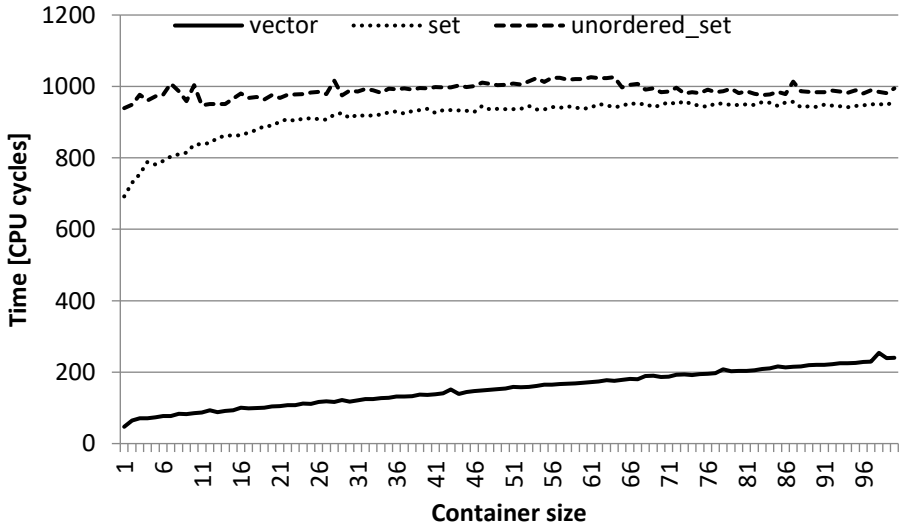

Figure 6 Vector performs better in this duplicate emplace scenario tested with warm cache.

We described the case when an element already is the container, but what about when it is not there? Figure 7 shows the results for a cold cache and Figure 8 for a warm cache. They both show that there is yet another thing we need to take into account – the growth of the container. A set is implemented as a tree, so it grows with each element added. An unordered set is keeping all elements in a list so it also grows with each element, but on top of that it manages a hash table with iterators to buckets of elements. The more elements we add, the greater the chance that more than one element will have the same hash value and will end up in the same bucket. Since the elements are not sorted in any way, the buckets are searched linearly. If we go back to Figure 1, we can notice that the average search time for an unordered set is slowly going up with the number of elements in the container up to 64 elements where it goes back down slightly. That slow increase is caused by more hash collisions which lead to more than one element in a bucket. This problem is mitigated by increasing the size of the hash table. When the table grows, all elements are assigned to new buckets. That process gets more expensive with more elements in the container.

Both a set and an unordered set container have to allocate space for new elements and at times the unordered set has to increase the size of the hash table. A vector on the other hand grows by 50% each time, so it has some extra space for new elements and it needs to allocate new space less often than the other two containers. When it grows though, it has to copy/move all elements to the new location. The measured function calls are the same as we saw back in Listing 2 but without duplicate data.

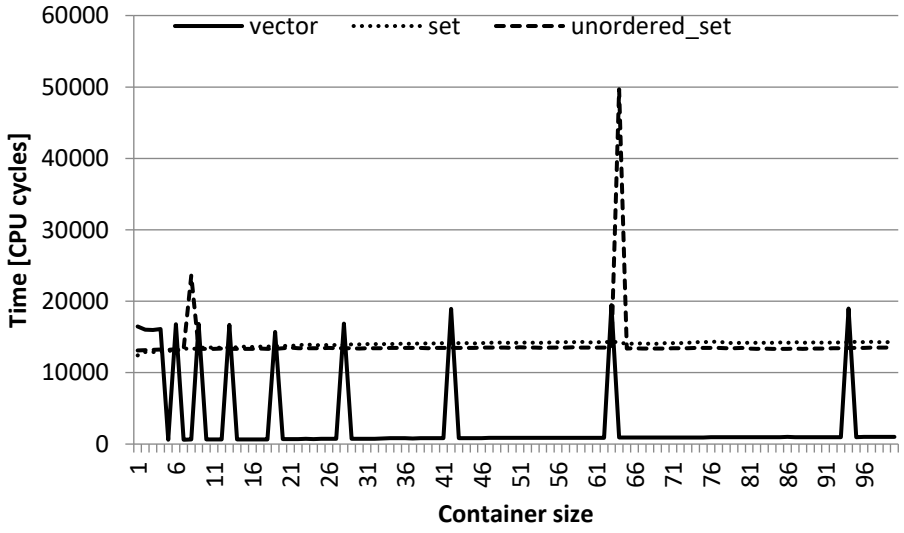

Figure 7 Allocations are way more expensive than search time when the cache is cold.

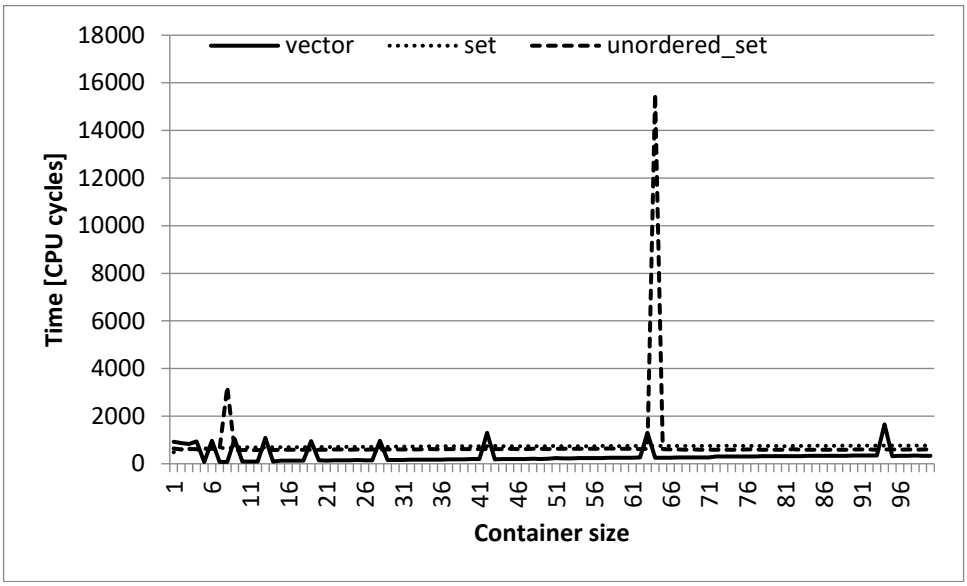

Figure 8 The re-hashing of the unordered set turned out to be the most expensive operation in this warm cache test.

4.4 Copy and iteration

The last two things we are going to look at are copy and iteration times. Let us say we have a system that takes a list of enemies and decides what enemy we want to shoot at. An enemy can be identified by a unique ID or by a pointer. The system will run on a worker thread, so it will create a copy of the input list of enemies. We don't want to process one enemy multiple times, so the entries should be unique. And since our new system is using fuzzy selection logic, it is going to iterate through all enemies. The std::set might look like an obvious choice when we try to implement such a system. So let us see how fast the copy of the input data would be.

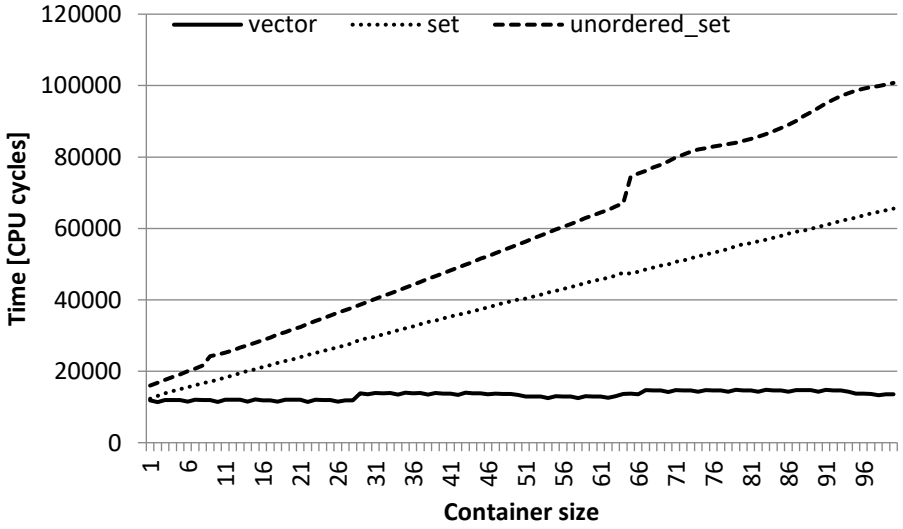

Figure 9 Cold cache copy test showing per element allocation costs.

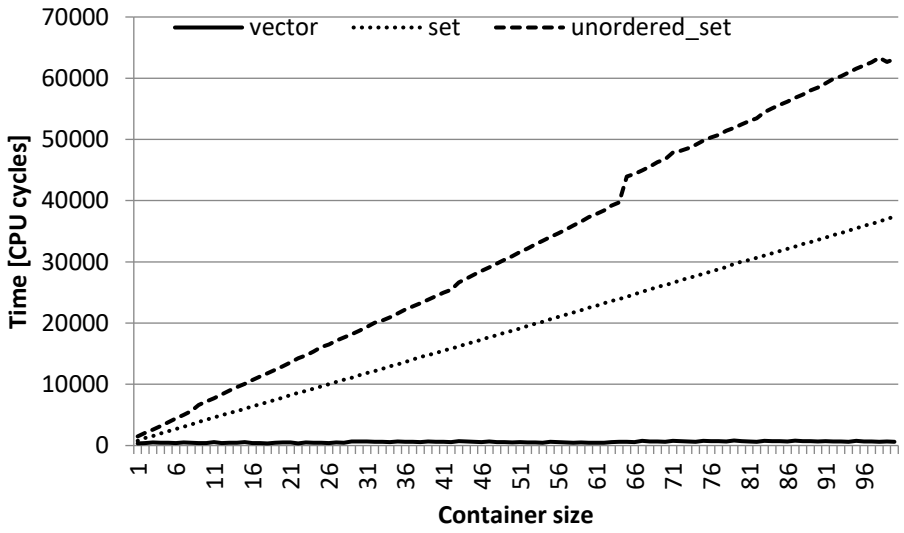

Figure 10 Warm cache copy test showing per element allocation costs.

Clearly the fact that a vector needs to do only one allocation instead of an allocation per element is a big advantage here. It is hard to see in Figure 9, but copying a vector with 100 elements is 10 times faster than copying an unordered set when the cache is cold. Figure 10 shows that copy is 100 times faster for a vector when the cache is warm!

How does iteration time look? We measured how long it takes to iterate over all elements in the container using code shown in Listing 4. Let us take a look at Figure 11 for the cold cache results and Figure 12 for the warm cache results.

The std::vector performs better in both cases. The difference between std::vector and std::unordered_set is not that big in the warm cache scenario shown in Figure 12. In the case of warm cache, we tried to keep the container in a cache and we also added elements in a tight loop. So it is very likely that most of the logically adjacent elements are also stored in adjacent physical memory. The std::vector always stores elements in a contiguous memory. The std::set performs the worst of these three containers even if we add the elements in a tight loop, or use a custom allocator to make sure elements are in one memory block. A set arranges elements into a sorted tree structure, so the iteration is jumping all over the memory and cache prediction fails.

The cold cache scenario, on the other hand, measures time needed for iteration over all elements when the data is not in cache and elements are scattered around the memory.

Listing 4 Function calls measured in the iteration test.

```
for(auto & e : container)
{
     auto temp = e;
}
```

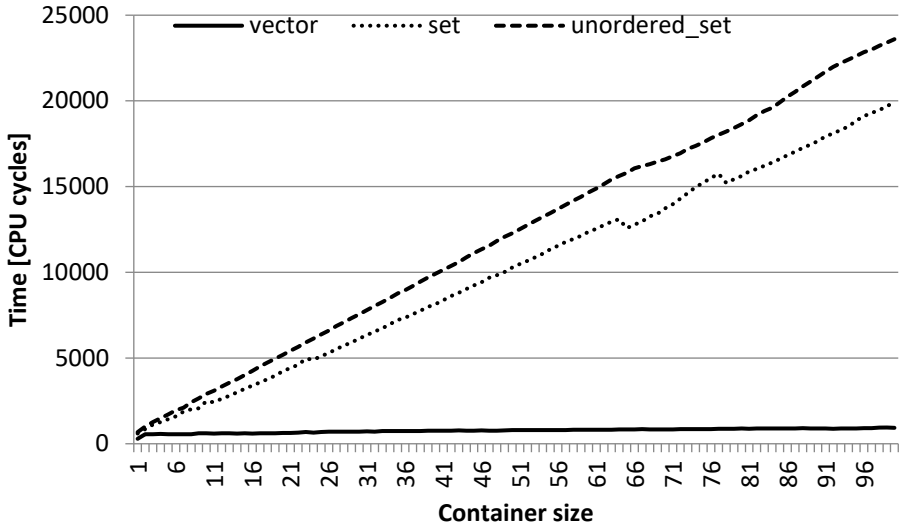

Figure 11 Iterations over elements that are not in cache and are scattered around memory is slow.

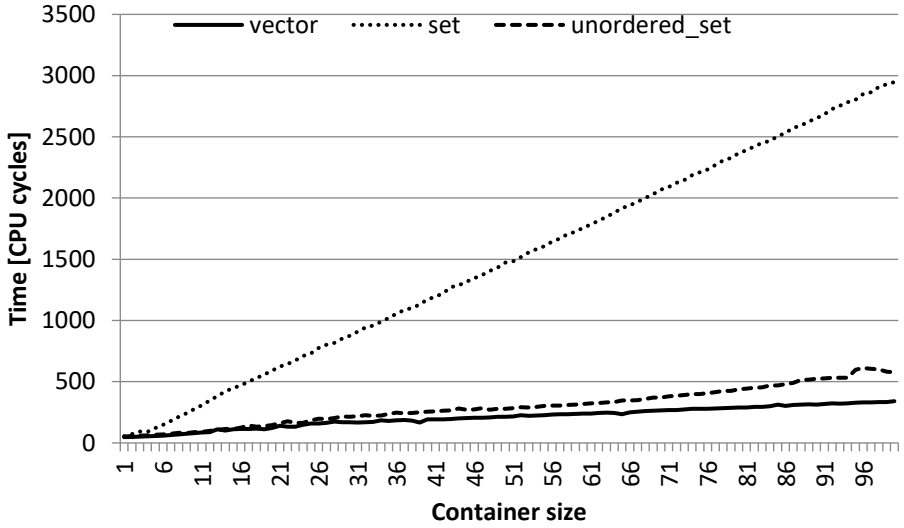

Figure 12 Iteration over contiguous memory is fastest.

5 Map and unordered_map

The std::map and std::unordered_map are also associative containers, but these hold a key-value pair. We could repeat all the tests we did in the previous section, but we would find out that the results are very similar. A vector would be winning in the same situations and we would be having a feeling of Déjà vu. What we are going to look at instead is what role the size of the value type plays.

We touched on the topic of CPU cache earlier so we know that data is loaded from main memory to cache for faster access. When we request some data, 128 bits are loaded (AMD 13) into the cache in one cycle, even if we need just one byte. But if we try to access the adjacent byte, it is already loaded and the access is much faster. So in theory, the smaller the stored item is, the faster the search should be in cases where the data are stored in contiguous memory.

We are going to create a simple wrapper on top of the std::vector container to store key-value pairs as shown in Listing 5. The test will show the time needed for finding one element in a container of size 100 in the cold cache scenario.

Listing 5 Custom associative container storing elements in std::vector.

```
template <typename K, typename V>
class C_VectorMap
{
public:
     typedef typename std::pair<K, V> storedType;
     typedef typename std::vector<storedType> storageType;
     typedef typename storageType::iterator iterator;

     void insert(storedType && p)
     {
          for(auto & r : m_Storage){
               if(r.first == p.first)
                    return;
          }
          m_Storage.emplace_back(
               std::forward<storedType>(p));
     }

     template <class... Args>
     void emplace(K key, Args&&... args)
     {
          for(auto & r : m_Storage){
               if(r.first == key)
                    return;
          }
          m_Storage.emplace_back(std::piecewise_contruct,
                    std::forward_as_tuple(key),
                    std::forward_as_tuple(
                         V(std::forward<Args>(args)…));
     }

     void erase(K key)
     {
          for(auto it = m_Storage.begin();
```

```
                it != m_Storage.end();
                ++it){
                if(it->first == key)      {
                     *it = std::move(m_Storage.back());
                     m_Storage.pop_back();
                     return;
                }
           }
     }

     iterator find(K key)
     {
          for(auto it = m_Storage.begin();
                it != m_Storage.end();
                ++it){
                if(it->first == key)
                     return it;
           }
           return m_Storage.end();
     }

     iterator begin() { return m_Storage.begin(); }
     iterator end() { return m_Storage.end(); }

private:
     storageType m_Storage;
};
```

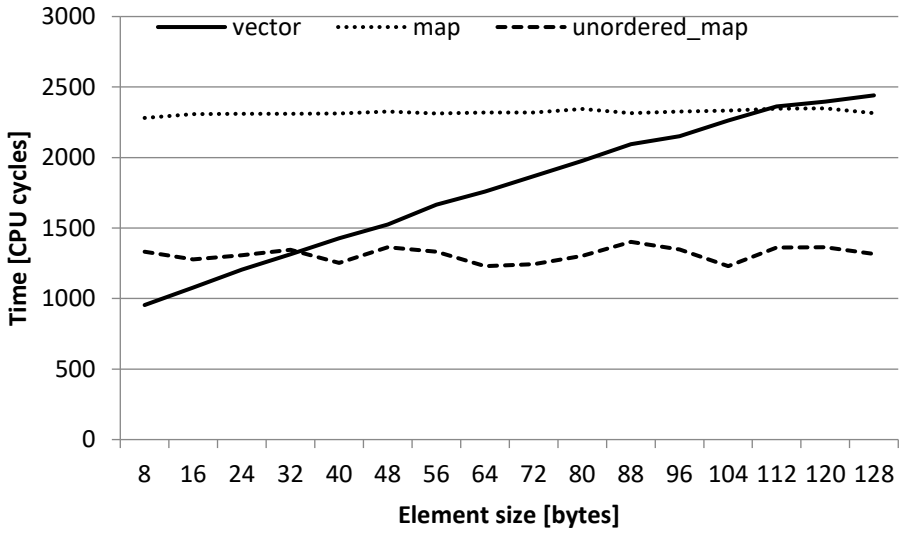

Figure 13 Bigger element size makes std::vector iteration slower.

The results in the Figure 13 show that with bigger element size fewer of them fit into cache. Element size influences the average search time of std::vector which leverages the benefits of contiguous memory iteration. The two other types of container are not influenced by the size of the element.

We can improve our custom implementation by using two std::vectors instead of one. The first one would store the keys and the second one the values. They would be in sync, so the first key would correspond to the first element value. When searching for the item, we would iterate over the key container only. When we find the correct key, we use its index to address the value in the elements container. Figure 14 shows how we removed the dependency on element size by this approach.

Using two std::vectors will make the final implementation more difficult if we decide to support the same interface as std::map. For example, the iterator cannot point to std::pair, because we are not storing the elements as that type. It would also mean that when we need to add/remove elements, we must do that at two places instead of one ultimately making operations like insert or erase slower.

Results in Figure 14 show one more thing. We used keys of size 32bits instead of 64bits and that made the std::map and std::unordered_map a tiny bit faster than in the previous test. This means that while those containers are not influenced by element size, they are influenced by the key size.

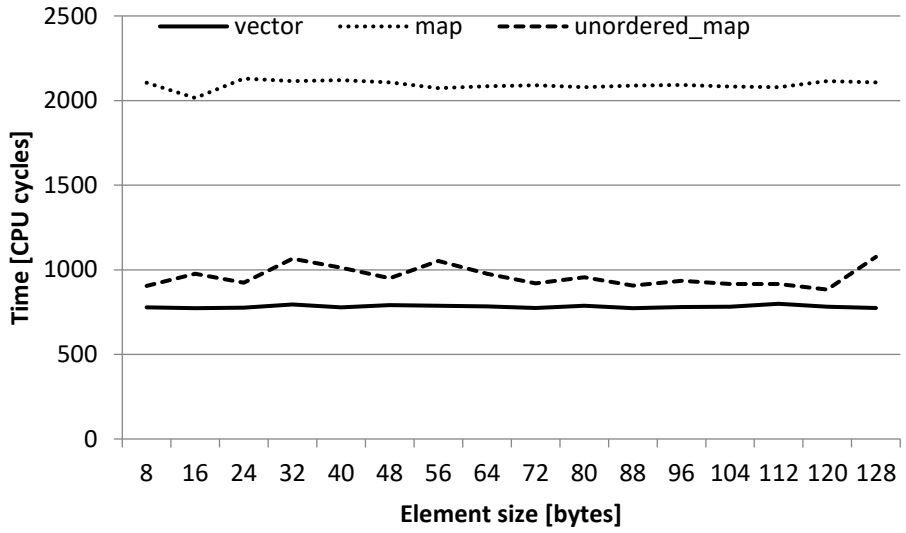

Figure 14 Using two std::vectors removes the dependency on element size.

6 What to avoid

When we decide to use STL containers in our game engine, there are a few things we need to keep in mind. As we saw, the search time complexity is one thing but the actual time is something else. Also the average search time is not the only metric that matters. Allocations are way more expensive than search times so we should choose the container type based on

the main usage. On top of that, each container comes with some memory overhead for each element that we need to bear in mind as well. Here are some tips on what to do and what to avoid.

- Rule number one should be, do not make any assumptions about performance and always measure it. We proved the assumption that "a container with time complexity O(1) has faster search time than O(*n*)" to be wrong under certain conditions. We also proved the importance of the cache when it comes to performance. So cache friendly code and data are going to be processed faster by the CPU (Nystrom 14). Making any assumptions about what is going to be in cache would be foolish and it should be measured as well.
- We should always reserve the size of the vector when we have an idea how many elements are going to be in there. When we take a look back to Figure 7 and Figure 8, we will see that an empty vector is allocating quite often when we start adding elements into it. Since the new size is 1.5 multiplier of the current size, it performs an allocation for each of the first 4 elements. Let us say we create a function which goes through all enemies and returns a vector of those with less than 50% of their health. Insertion of the results into the output vector can be far more expensive than anything else we do in the function when we do not reserve the space in the vector in advance. An even better option would be if we do not allocate any dynamic memory at all, in case the output vector is temporary in the caller function. We will show one of the solutions in the next section.
- We should avoid using associative containers when we do not really need them. Let us say we have a manager for AI components. Each component can be registered into the manager only once and all the registered components are updated every frame. As we know, contiguous memory is the best choice for iterations and that would be the most frequent operation in this case. One component cannot register into the system twice, but that does not mean that we have to use std::set or std::unordered_set for holding the list of unique pointers to the registered component. We can implement this using the vector and make sure on the caller side that we don't register one component twice. The register function itself can check that the component is not already there and assert if it is on debug builds only, for example. The unregister function needs to find the element for removal even in release build, but similar to the register function, it can have the branch in case the element is not there, and assert on develop builds only as well.
- The emplace function introduced in C++11 is not always faster than the insert function. Consider using emplace instead of insert, but remember that emplace is creating the new element in-place, passing the function parameters into the constructor.
- We can greatly improve the performance by using custom allocators. We said many times that the allocations are very expensive especially when the cache is cold, and that contiguous memory is more cache friendly. Therefore, we can improve the performance by creating custom allocators. We can have an allocator per system, for example, so the AI objects would be in one part of memory and, let us say physics objects, would be in other parts of memory. We

can also have an allocator for temporary objects. That would be for dynamically allocated objects that live for one frame only. Such an allocator would defer all deallocation to the end of the frame when it returns all the allocated memory to the global allocator at once (Gregory 09).

- We should avoid copying containers, especially when it is implemented as some kind of linked list. This includes a set, which is a tree structure, and an unordered set, which uses a list for storing its elements.
- We should never use std::find on an associative container. This algorithm performs a linear search on the range of elements provided by the first and last iterators as parameters. If we do that on a std::set, for example, we would iterate linearly through the elements until we find the one we are looking for. That would completely ignore the tree structure which guarantees the O(log *n*) time complexity and we would do the slow iteration instead.

7 Conclusion

While it is possible to ship a game using the STL containers in our game engine, we should consider all pros and cons before going that way. The STL containers are well tested and proven to work, they have great documentation and a wide user base, so we can find answers to our problems on the Internet easily. Another plus is that they are standard, so when you hire a new programmer, they should already know how they work. Last, but not least, the standard library offers other stuff as well, for example the algorithms, which will improve your productivity as well. On the other hand, we have seen that a contiguous memory container can give us a better performance and smaller memory footprint in certain cases which can be a very valid reason for implementing custom containers.

8 References

[AMD 13] AMD. 2013. Software Optimization Guide for AMD Family 16h Processors http://support.amd.com/TechDocs/52128_16h_Software_Opt_Guide.zip (accessed March 10, 2018)

[Celis 86] Pedro Celis. 1986. Robin Hood Hashing. PhD diss., University of Waterloo, Ontario.

[Cormen 90] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. 1990. *Introduction to Algorithms*. Cambridge: MIT Press.

[Gregory 09] Jason Gregory, Jeff Lander, Matt Whiting. 2009. *Game Engine Architecture.* Boca Raton: Taylor & Francis.

[Hruska 17] Joel Hruska. 2017. How L1 and L2 CPU Caches Work, and Why They’re an Essential Part of Modern Chips. https://www.extremetech.com/extreme/188776-how-l1-and-

l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips (accessed March 10, 2018)

[IGN 16a] IGN. 2016. PlayStation 4 Hardware Specs. http://www.ign.com/wikis/playstation-4/PlayStation_4_Hardware_Specs (accessed March 10, 2018)

[IGN 16b] IGN. 2016. Xbox One Hardware Specs. http://www.ign.com/wikis/xbox-one/Xbox_One_Hardware_Specs (accessed March 10, 2018)

[Nystrom 14] Robert Nystrom. 2014. *Game Programming Patterns.* Genever Benning.

[Sylvan 13] Sebastian Sylvan. Robin Hood Hashing should be your default Hash Table implementation. https://www.sebastiansylvan.com/post/robin-hood-hashing-should-be-your-default-hash-table-implementation/ (accessed March 10, 2018)