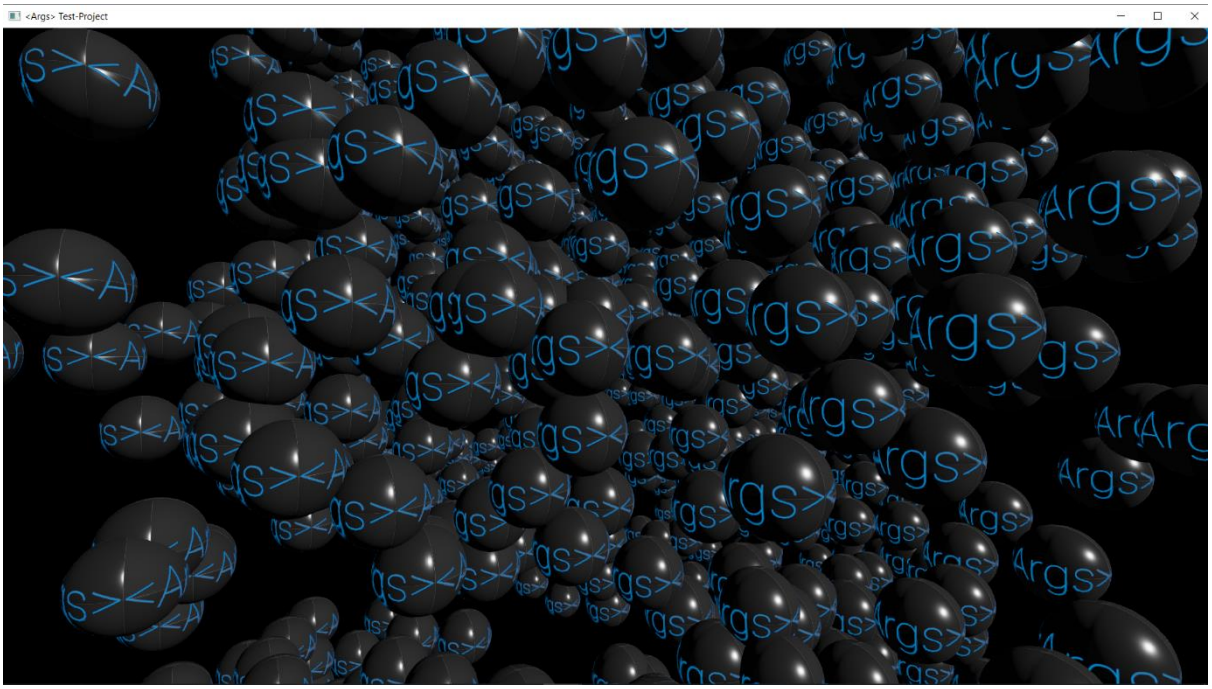
Research: Advanced Tools

Entity component system optimisations



Conducted by Glyn Marcus Leine
Date 17 04 2020
Archive title Optimisations to the Args engine core systems

Table of Contents

Proposition one-liner

I will research different ways to optimise the entity component system of the Args game engine.

Proposition description

The proposed topic of research is optimisations that could be made to the Args game engine, in particular to its entity component system. The ECS shapes a big part of the core of the Args game engine, however, due to time constraints, a lot of it was rushed during our last project. In all of that rushing, a lot of shortcuts were made for the sake of implementation time and simplicity. Removing those shortcuts and finding better, more permanent solutions would be highly beneficial to the engine and thus future use of the engine.

Justification

Entity component system (ECS) is a fairly young architectural pattern that has been on the rise with the data-oriented programming paradigm. It has been dubbed the future of game engines by several sources. As Tim Johansson from Unity describes in their DOTS GDC talk: “performance by default” (Unity Technologies, 2018). And as the Youtuber Brackeys says in their video about the Unity ECS: “Extremely performant code, easier to read, easier to reuse code” (Brackeys, 2018).

All this talk about performance comes from the data-oriented nature of ECS’. This is important because over the years that computers have evolved the CPU has left memory in the dust. Thus, your CPU spends a lot of time waiting on memory. The way the CPU tries to minimalize this wait time is by trying to predict what memory it might need in the future and load it into faster memory (cache) beforehand.

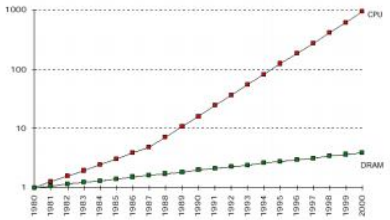


Fig. 1. Processor-memory performance gap: starting in the 1980 performance, the microprocessor and memory performance over the years [13]

The Gap between Processor and Memory Speeds (Carvalho, 2002)

This prediction method, however, is not flawless and thus sometimes the CPU still needs to access memory that was not loaded into cache. This is known as a cache miss. These cache misses could be prevented by decreasing data sizes (chopping up into components) and by writing more easily predictable code (storing data contiguously). An ECS does both of these next to also reducing the number of unnecessary items to be updating (empty objects).

References

Brackeys. (2018, June 3). New way of CODING in Unity! ECS Tutorial. https://www.youtube.com/watch?v=_U9wRgQyy6s&feature=emb_title

Carvalho, C. (2002). The Gap between Processor and Memory Speeds. https://pdfs.semanticscholar.org/6ebe/c8701893a6770eb0e19a0d4a732852c86256.pdf

Unity Technologies. (2018, March 23). Job System & Entity Component System. https://www.gdcvault.com/play/1024839/Job-System-Entity-Component-System

Proposition details

The above-mentioned shortcuts are not well documented due to those same time constraints also mentioned. So, the likely hood of finding more overlooked shortcuts in the eventual research is high. However, these are the known issues that need research and solutions:

STL set vs custom sparse set class

In order to save time, the ECS uses a sparse set like data structures in critical areas and uses std::set in less critical places. This also means that an abstraction for the sparse set data structure was never made for re-use throughout the engine.

The difference between a sparse set and the STL set should be quite considerable due to the STL set being node-based and thus also non-contiguous. Whilst the sparse set data structure has similar features but is based on a packed data array that is thus always kept contiguous.

This packed array can also be replaced by the STL vector to create a dynamically growing sparse set. And in combination with object pooling, the set would also prevent reallocations once the set has grown to its largest size.

Remove dependency on strings

Another time-saving shortcut we made was using strings everywhere throughout the engine as keys to store data to. This was beneficial for the deserialization systems when loading from text-based files like JSON and XML but has high performance costs.

The main performance cost comes in the shape of hashing functions and can sometimes take up 95% of the frame time in debug mode (release makes some optimisations that drastically reduces the cost). In some cases, this makes the engine almost unrunnable in debug mode. The hashing function will be difficult to completely remove unless we can find a way to give datatypes a numerical id that is consistent between runs and compiles even after adding new features. This id would also need to be replicable in a separate application for tooling reasons.

This id method would be ultimate but possibly slightly out of scope for now. However, there's a possible solution in the meantime: to remove the dependency on strings in all areas besides the deserialization. This would likely also be the way we could create the replicable id, by generating a unique numerical value from the string representation. No matter how "dirty" it might seem, it would work.

Research results

STL set vs custom sparse set class

Testing methodology

For the following tests, the entity component system was temporarily deoptimized to update the entity lists with every operation, not only the relevant ones.

The engine was then loaded with a scene of a thousand entities being rendered and lit. Each one of the entities would have a component queried, a component added, and a component removed, 10 times per frame using the entity component system.

These operations were measured individually. The data was then processed, extracting the lowest recorded time, the highest recorded time and the average time of each operation.

This test was conducted over 5 minutes of run time and validated over several runs for both debug mode and release. In between each run, the computer was left to settle and cool down so that each run would have a comparable starting state. No other programs besides visual studio were allowed to run in the background.

Test subjects

The tested data structures consist of the STL set and unordered set and two variants of a custom sparse set implementation. For both sparse sets, the dense "array" comes in the form of the STL vector. However, the sparse "array" comes in the form of either the STL vector or the deque.

It shouldn't be much of an issue for the sparse "array" to be a node-based data structure such as the STL deque due to the sparse "array" not having the same necessity for being contiguous as the dense "array".

Hypothesis

In the tables below, the complexity of several operations on certain data structures are displayed. These complexities are sourced from www.cppreference.com, except for the complexity of the sparse set, which is based on my own research.

Based on these complexities the sparse set should turn out on top.

Lookup

structure	Average case	Worst case
vector	O(N/2)	O(N)
set	O(log(N))	O(log(N))
unordered_set	O(1)	O(N)
sparse set	O(1)	O(1)

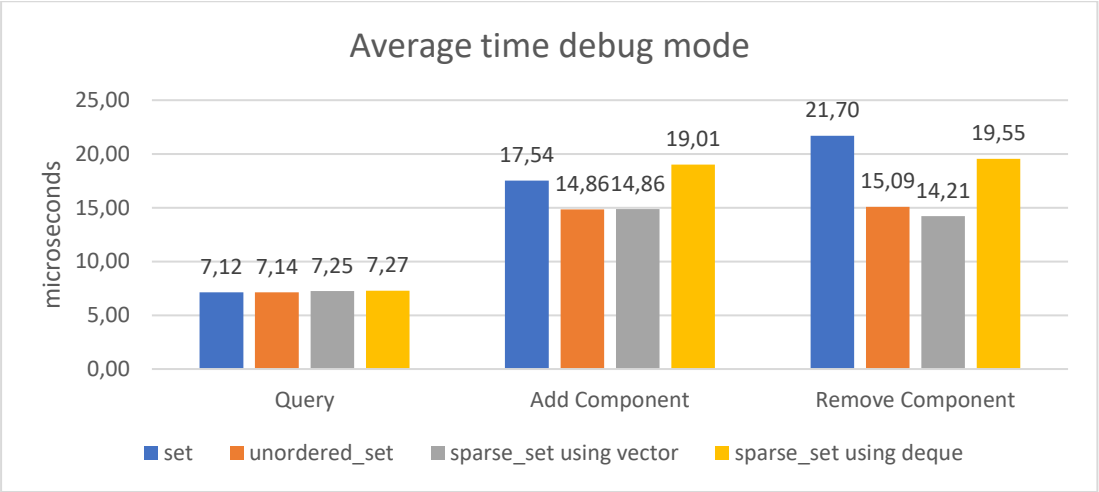
Insertion

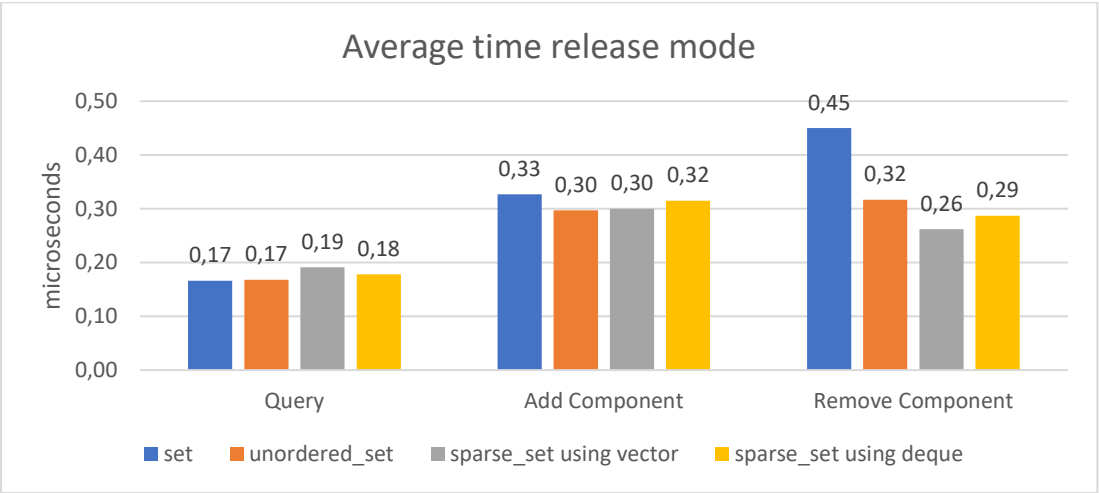
structure	Average case	Worst case
vector	O(1)	O(1)
set	O(log(N))	O(log(N))
unordered_set	O(1)	O(N)
sparse set	O(1)	O(1)

Removal

structure	Average case	Worst case
vector	O(2N)	O(2N)
set	O(1)	O(1)
unordered_set	O(1)	O(N)
sparse set	O(1)	O(1)

Test results and analysis (Averages)





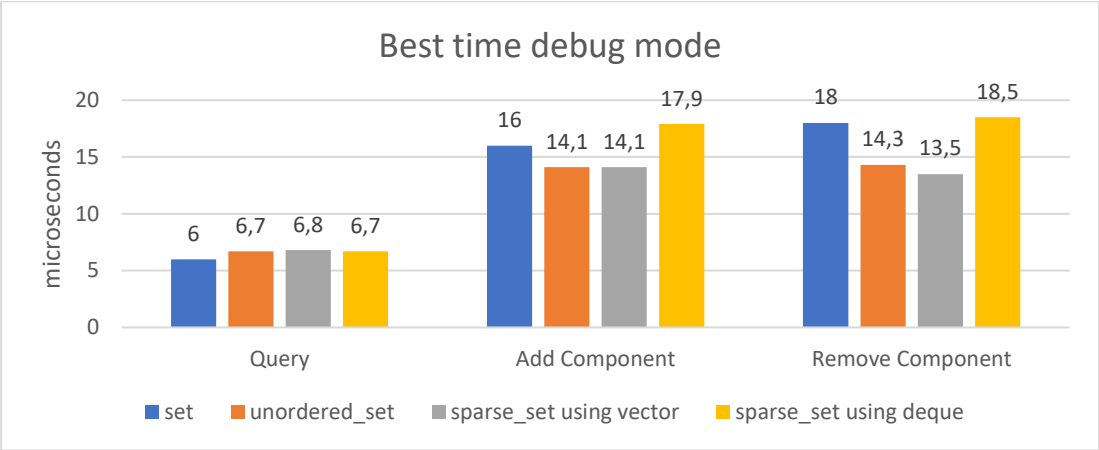
The recorded averages show that going to std::unordered_set instead of std::set already has an advantage. Although the difference is a mere few microseconds the difference is enough to make it noticeable in the average frame rates.

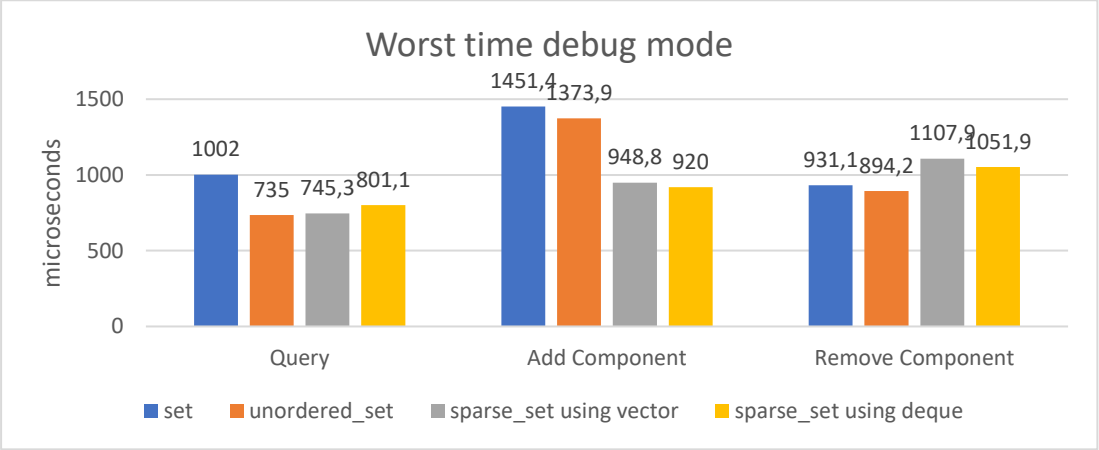
Frame Rate	Set	Unordered set	Sparse vector	Sparse deque
Debug	1,94fps/516,7ms	2,38fps/420,8ms	2,44fps/410,6ms	1,94fps/516,5ms
Release	86,61fps/11,5ms	101,81fps/9,8ms	105,53fps/9,5ms	102,49fps/9,8ms

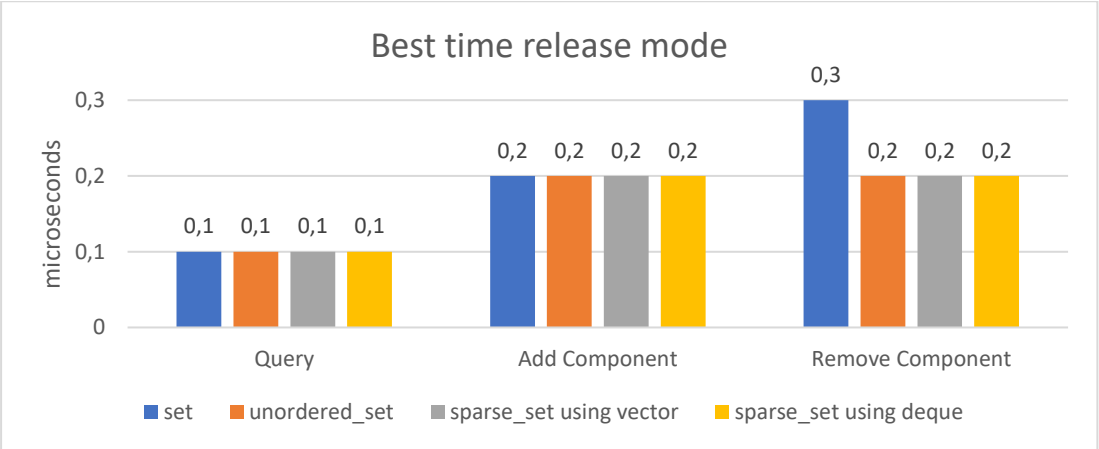
An interesting thing to notice is that despite the sparse set having a lower theoretical complexity, the deque version of the sparse_set has very little impact compared to the STL set in debug mode but surpasses the unordered_set in release mode. Besides that, the unordered_set performs ever so slightly poorer than the vector-based sparse_set despite the supposed average-case complexity being the same.

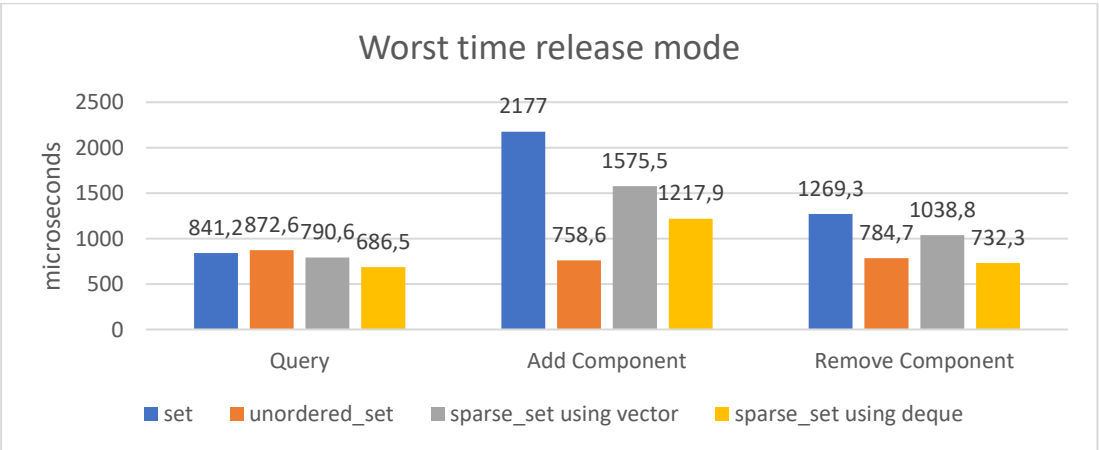
Despite a definite winner not being so obvious, it is clear that the STL set and the deque sparse_set are the worst two of the four structures. Between the other two, it seems to be a matter of use-case, a more unbalanced testing methodology could severely favour either one of the two.

Test results and analysis (Best and Worst)

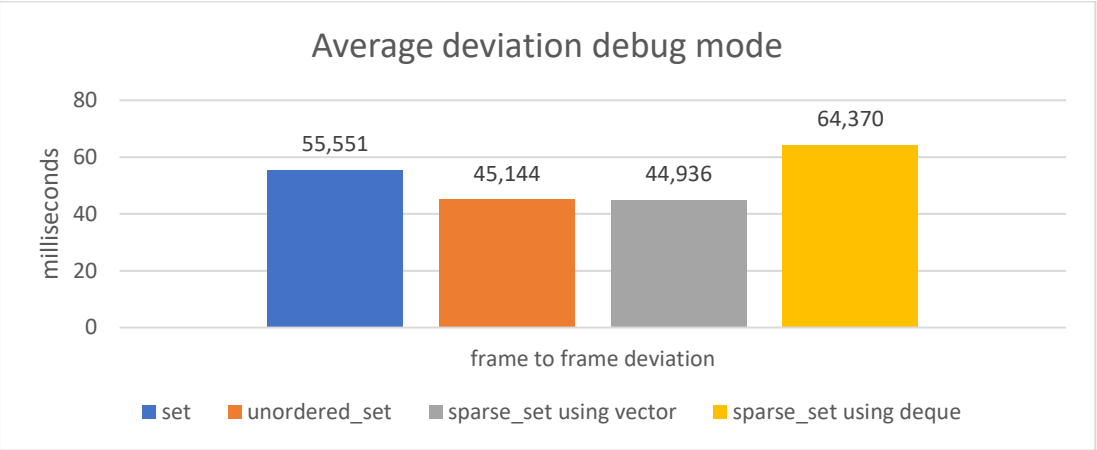


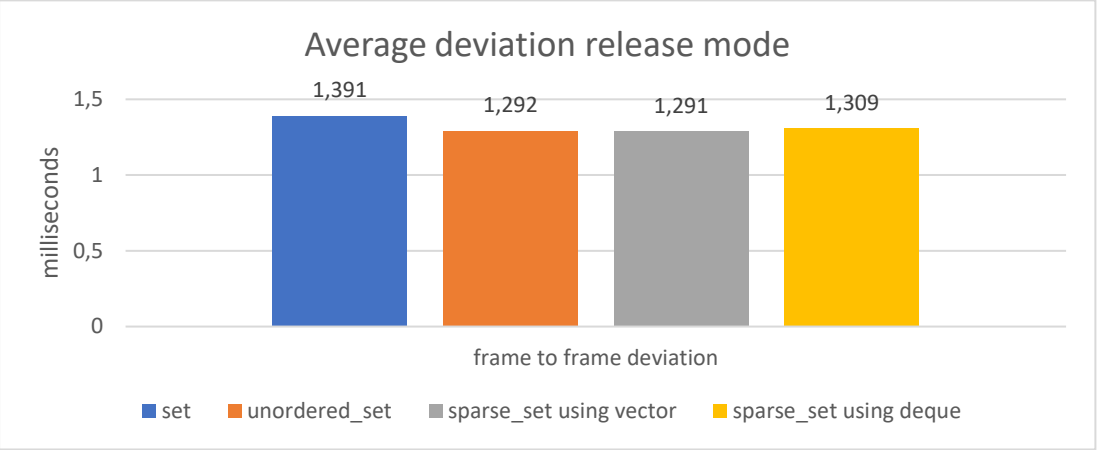






The charts above show the best and worst recorded times of each operation in both debug and release mode. The reasoning behind this data recording was to see the stability of the structures. If a data structure managed to get a very high average frame rate but also introduces tremendous frame spikes, then the structure should still be deemed unfit since stability is of such importance to game engines. Together with these best and worst cases, the average frame to frame deviation was also recorded.





It further shows the unreliability of the deque based sparse_set, and the STL set despite the deviation settling slightly in release mode.

With the average deviation combined with the best and worst recorded cases, we can also guestimate the frequency of the frame spikes. How often do we get closer to that worst recorded case?

The unordered_set, for instance, has a particularly bad wort recorded time in component addition in debug mode. However, the average frame to frame deviation is on par with the vector sparse_set. This shows that the spikes of the unordered_set are less frequent or at least less frequently as severe as the worst recorded case.

Interestingly the unordered_set has better stability in release mode than the vector sparse_set, despite the opposite being true in debug mode.

Conclusion

Due to the slightly better average performance in both debug and release mode the Args engine will continue forth with the vector-based sparse_set. Despite the worst-case recording of the vector sparse_set being worse than that of the unordered_set, the average

deviation is better. This also alludes to the spike not being very frequent and possibly even only being a one-time thing. These stability tests could use another round of testing with a different methodology of capturing and processing the measured data. A frequently used measure in benchmarking are 1% lows/highs, averaging the 1% best and worst recorded data.

Furthermore, the sparse_set, in both vector and deque forms, can evolve more over time as better ways to write the implementation are found and the structure gets more optimised. Next to that, the sparse_set can now also be implemented in more critical areas of use which were using a temporary sparse set implementation up till now. A keyed variation of the sparse_set might be handy for this and is the immediate concurrent area of focus for the Args engine ECS.

Remove dependency on strings

Testing methodology

For the these tests the state of the engine is quite similar to the state of the STL set tests in the previous topic. One major difference however is that the temporary deoptimizations have been removed. So, the string-based case is the engine using the old string-based communication system and the STL set as the engine used to do before this research was conducted. The difference between the string-based and non-string-based test is thus only the communication channel through strings or unique ID's.

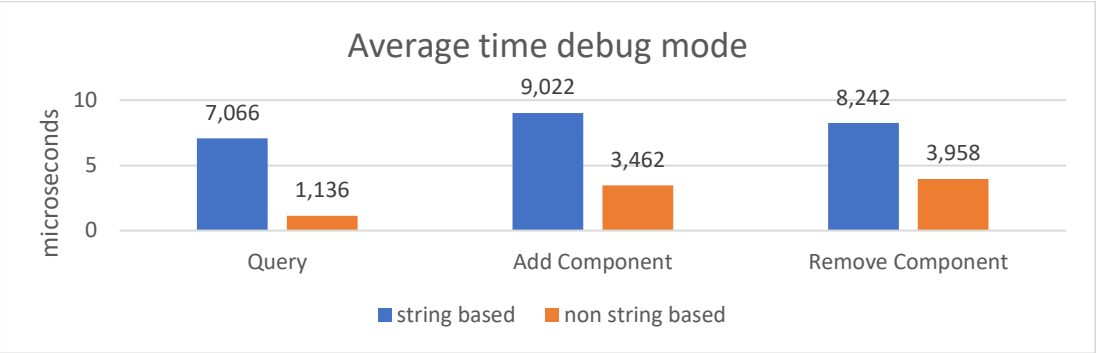
As for the improvement being made in the non-string-based approach.

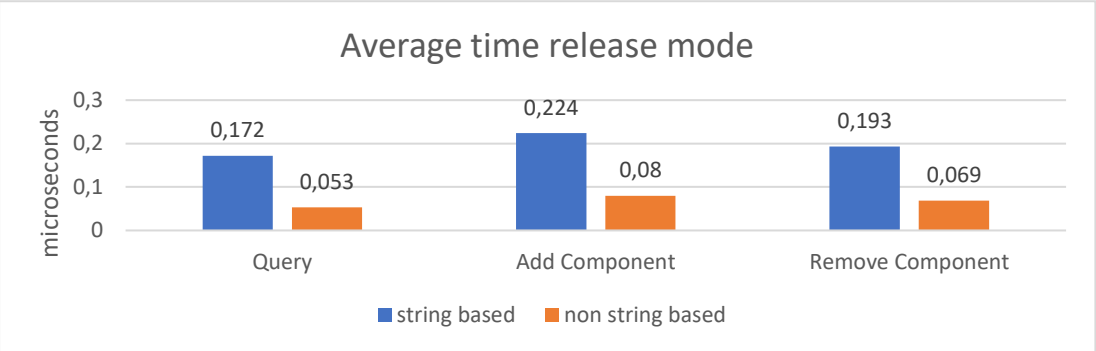
All use of string within the engine around component creation, querying and destruction has been replaced with a runtime unique numerical id. In order to retain compatibility with external sources each component type will get an external id which is run and compile independent. For the very slight instance strings are still needed a function to fetch/calculate the external id from the string is provided. The external id can be directly mapped to the internal id.

Hypothesis

Since strings take quite a long time to get hashed using strings to index into data structures is a very costly operation. The complexity of hashing the string is linear to the length of the string. Doing this multiple times per update of a component is likely one of the largest issues the engine has as of writing this paper. Replacing all the strings for unsigned integers and reducing all the strings only to the area of registering the components could greatly improve performance.

Test results and analysis

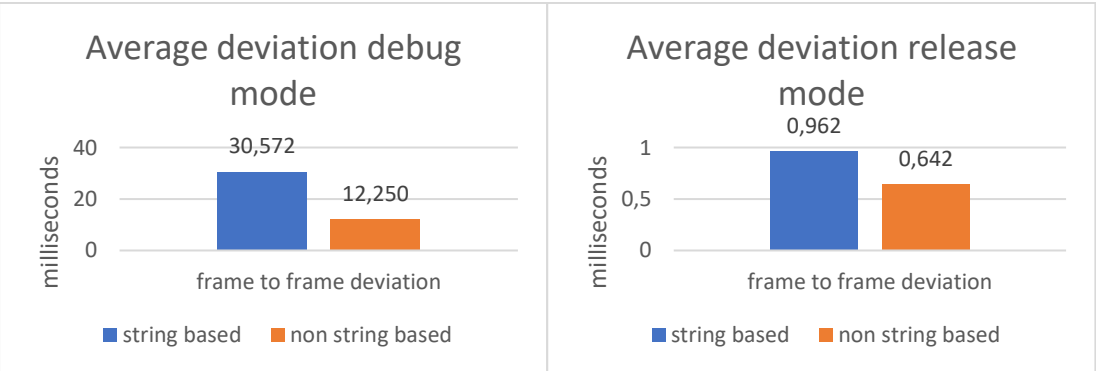




As expected the time per operation has drastically decreased when switching over to the non-string-based approach. This is also greatly reflected in the average frame rates.

Frame Rate	String based	Non string based
Debug	3,498 fps/285,8ms	8,787 fps/113,8ms
Release	126,378 fps/7,9ms	255,216 fps/3,9ms

Besides the increased average frame rates the new approach also comes with an immense stability increase. The average frame to frame deviation is reduced a lot, which is especially noticeable in debug mode.



Conclusion

There is more than enough evidence showing that the non-string-based approach is superior to the string-based approach in every single way. Thus, the Args engine will also continue onward with this approach.