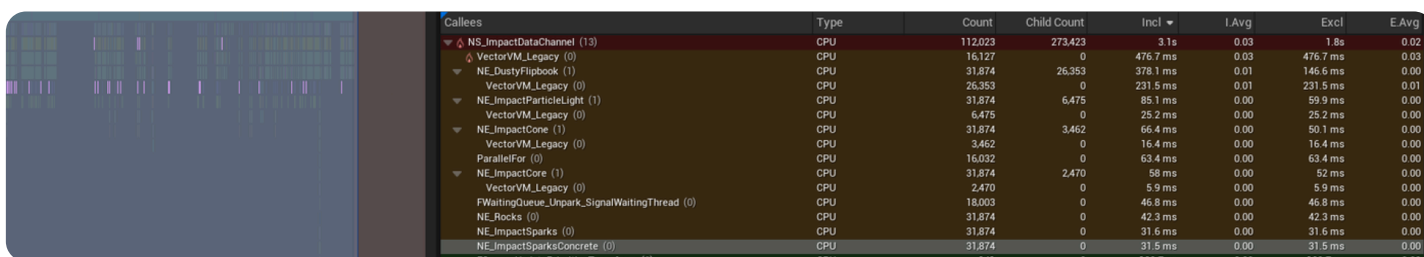


Systems as a Service

Learn about two different approaches used by the Lyra project to dynamically spawn weapon impacts into Niagara systems.



Calles	Type	Count	Child Count	Incl	L.Avg	Excl	E.Avg
NS_ImpactDataChannel (13)	CPU	112,023	273,423	3.1s	0.03	1.8s	0.02
VectorVM_Legacy (0)	CPU	16,127	0	476.7 ms	0.03	476.7 ms	0.03
NE_DustyFlipbook (1)	CPU	31,874	26,353	378.1 ms	0.01	146.6 ms	0.00
VectorVM_Legacy (0)	CPU	26,353	0	231.5 ms	0.01	231.5 ms	0.01
NE_ImpactParticleLight (1)	CPU	31,874	6,475	85.1 ms	0.00	59.9 ms	0.00
VectorVM_Legacy (0)	CPU	6,475	0	25.2 ms	0.00	25.2 ms	0.00
NE_ImpactCone (1)	CPU	31,874	3,462	66.4 ms	0.00	50.1 ms	0.00
VectorVM_Legacy (0)	CPU	3,462	0	16.4 ms	0.00	16.4 ms	0.00
ParallelFor (0)	CPU	16,032	0	63.4 ms	0.00	63.4 ms	0.00
NE_ImpactCore (1)	CPU	31,874	2,470	58 ms	0.00	52 ms	0.00
VectorVM_Legacy (0)	CPU	2,470	0	5.9 ms	0.00	5.9 ms	0.00
FWaitingQueue_Unpark_SignalWaitingThread (0)	CPU	18,003	0	46.8 ms	0.00	46.8 ms	0.00
NE_Rocks (0)	CPU	31,874	0	42.3 ms	0.00	42.3 ms	0.00
NE_ImpactSparks (0)	CPU	31,874	0	31.6 ms	0.00	31.6 ms	0.00
NE_ImpactSparksConcrete (0)	CPU	31,874	0	31.5 ms	0.00	31.5 ms	0.00
EScale_UpdatePrimitiveTransform (0)	CPU	141	0	180.7 us	0.00	180.7 us	0.00

Overview

Before 5.4 the APIs provided for spawning Niagara systems mostly supported a one-to-one relationship between system instances and gameplay events/external triggers. The Niagara systems had little in the way of allowing external code to pass in information on when to spawn particles and how to initialize them outside of activation and moving the component. This presents a problem if many instances of the same system need to be spawned in a short period of time, as the activation cost, and instance counts can cause major performance issues. Conceptually Systems as a Service (SaaS) takes a many-to-one approach between external triggers and system instances. That is to say that one system instance can handle spawning particles from multiple triggers without needing to be reactivated, avoiding the costs associated with activation and ticking multiple instances.

User parameters, and the Array Data Interfaces made it possible for advanced users to “inject” particles into an existing system, but there were limitations to this workflow, and it required lots of manual data management when building the arrays. Additionally there’s a trade off between the size of a SaaS System Instance, and instance culling that the array-based approach has to manage manually. As an extreme example, if all particle effects in a project were in the same system, in most cases the particles offscreen would greatly outnumber those onscreen, but their simulations would still be run, since the system itself would always be active.

To address the issue more generally, and make the setup easier, we developed Niagara Data Channels, which serve as a generic way for passing data between Niagara systems and gameplay. The main focus has been supporting the SaaS use case, but there are other use cases for Data Channels that we won’t cover here. We created a specific type of Data Channel, the Islands Data Channel, that handles subdividing a level, and spawning instances of a system into those “islands” for data that’s written within its bounds. The Data Channel

asset defines the data it will contain, and what systems it will spawn, and handles everything automatically when data is written to it.

We are going to cover both approaches in this guide both to give options and a comparison between the two, and to show an upgrade path. The Lyra project used array based spawning in previous versions, and for 5.4 we updated its impacts to support using Niagara Data Channels, so we will be using Lyra as both example cases.

When sending payload data, either with arrays or Data Channels, the general flow will be writing the data, usually from gameplay code, spawning particles based on that data in the Niagara system, and reading back that data to initialize the particle.

User Parameter SaaS

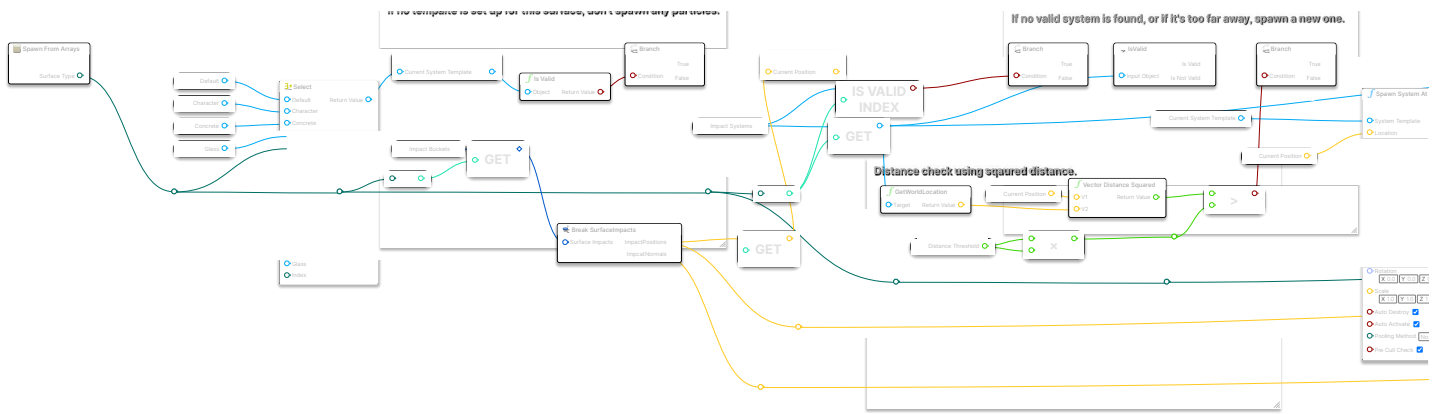
At its simplest a User Parameter can be used to change the spawn rate, or burst count of a particle system, which could be used to “add” particles to an existing system. For things like muzzle flashes that are attached to a single component and don’t need to respond to a projectile’s direction, this could be a viable setup, with limited need for managing the Niagara component’s lifetime or position. For more complex use cases like impacts and tracers, array based spawning can serve as a way of passing payloads of data to the system. Prior to 5.4 this was the only strategy employed for Lyra’s impacts for Concrete and Glass surfaces, and is still used for tracers. For 5.4 it’s still the default behavior, but can now be toggled to use the experimental data channels feature.

Lyra Impacts from Arrays

Lyra impacts are managed by the B_WeaponImpacts class. An instance of this is created in the class B_Weapon from the “Fire” custom event. Having a separate object owning impacts solved an issue where impacts would despawn when switching weapons. B_WeaponImpacts has a default system template for each surface type, and will spawn that system when a new impact on that surface occurs. If an existing system for that surface is close enough to a new impact, that system instance will be used instead of spawning a new one. Once the impact system instance is found or created, the relevant data (e.g. Impact Positions, Impact Normals, Muzzle Position) is written to the corresponding user parameters on that system.

Writing Data

Since the payload data is already an array, it can be written to the system directly using the functions in UNiagaraDataInterfaceArrayFunctionLibrary, e.g. Set Niagara Array Vector, Set Niagara Array Float, Set Niagara Array Position. Each surface has a different system associated with it, so before writing the data it's easiest to sort it by the surface type.



BLUEPRINT

Renderer by [Rancoud](#)

Copy code

Spawning

There are two Niagara system assets that support array spawning for Lyra impacts, NS_ImpactGlass and NS_ImpactConcrete. These systems are comprised of emitters based on the same parent assets that are then customized in the systems. These emitters use Spawn Burst Instantaneous modules to spawn their particles and the custom module NM_ImpactSpawnAttributes to read the data from the array user parameters. Some of the spawn modules have a fixed size, and some use the passed in User.NumberOfHits parameter to modulate the number of particles spawned.

Reading from Arrays

The Impact Spawn Attributes module is a bit more involved. It uses the particle's execution index, along with a start offset and the number of hits to index into the provided arrays. The start offset was originally used because all data from all impact surface types were bundled into the same array. The start offset was the index where a particular surface's data started, so each system would read from the offset that corresponded to its surface type. The Number of hits was used as a modulus to loop the particles back around if there were more particles than impact positions. The module then used this data to calculate values that were used elsewhere in the simulation, for example the impact normal, tangent and particle position.

With the refactor to support data channels, the arrays for this data were separated into structs based on surface type. This makes it easier to support different spawning strategies for different surface types, and eliminates

the need for the start offset.

With the way these systems were set-up they looped once, and never spawned more than one burst of particles per activation. This avoided any issues where the array data may have been out of date. Values can be read and removed from the end of the array to clear the data as it is read with the Remove Last Elem node. This approach doesn't work for these impact systems as the data needs to be read multiple times as there are multiple emitters. The values could be cleared externally or in the particle simulations, but that could lead to race conditions. Re-activating the system with fresh data is an acceptable compromise, though there is overhead to re-activating the system.

Data Channel SaaS

Data channels handle many aspects of the SaaS approach automatically, including writing data to the correct systems, spawning and placing those systems, and system lifetime management. This simplifies the use case, and optimizes some of these steps to offer better performance than an array based approach. For an intro tutorial to Niagara Data Channels follow the link below. To summarize: creating a Niagara Data Channel asset defines the structure of the data being sent to Niagara systems, and in the case of the Islands Data Channels, defines what systems to spawn, and how to spawn them when data is written to that data channel. The system then defines the behavior for spawning particles and reading that data to be used within the wider simulation. Data channels provide nodes to facilitate conditional spawning, and spawn ranges to allow for spawning a

-----●-----

The Data Channel specific behavior is handled in the function Spawn from Data Channels, which initiates a data channel write based on the number of impacts and the first position written to the array. This data is passed in to the search params, which are used to find the correct island to use for the current batch of writes, which determines the system written to. In the Lyra example it is assumed that the impacts will be close to the first position written, though it's possible to have impacts spread further apart, for example the shotgun spread could collide partially with close geometry, and partially with geometry far away.

Once the batch has been initiated with "Write to Niagara Data Channel (Batch)", the Impact Positions array is iterated over to write all the impact data to the data channel. The batched writes will accumulate the data written in this loop, and then publish it to the Niagara simulation when finished. For the data channels the two

systems NS_ImpactGlass and NS_ImpactConcrete were merged into one system NS_ImpactDataChannel. The emitters from the previous systems were versioned and updated to spawn from the data channel in their emitter update script, and read from it in their particle spawn scripts.

Fullscreen Reset Graph Zoom -10



BLUEPRINT

Renderer by [Rancoud](#)

Particles are spawned from data channels in an emitter module by calling `Spawn Conditional` on a `Data Channel Reader` that has been initialized to read from the corresponding data channel. For Lyra impacts this is mainly handled by the `NM_ImpactSpawnDataChannel` module.

The spawn logic is handled in `UNiagaraDataInterfaceDataChannelRead::SpawnConditional`, which checks to see if there is any data to be read from the data channel, then creates a spawn info for each instance of data. There are other mechanisms for conditional spawning, with the ability to accumulate or override the number of particles to spawn for each spawn info. The conditional logic can be used to control things like only spawning from specific surfaces, and the accumulation overrides can be used to chain multiple spawn calls together. For example, having a system that spawns extra particles, but only for a given surface.

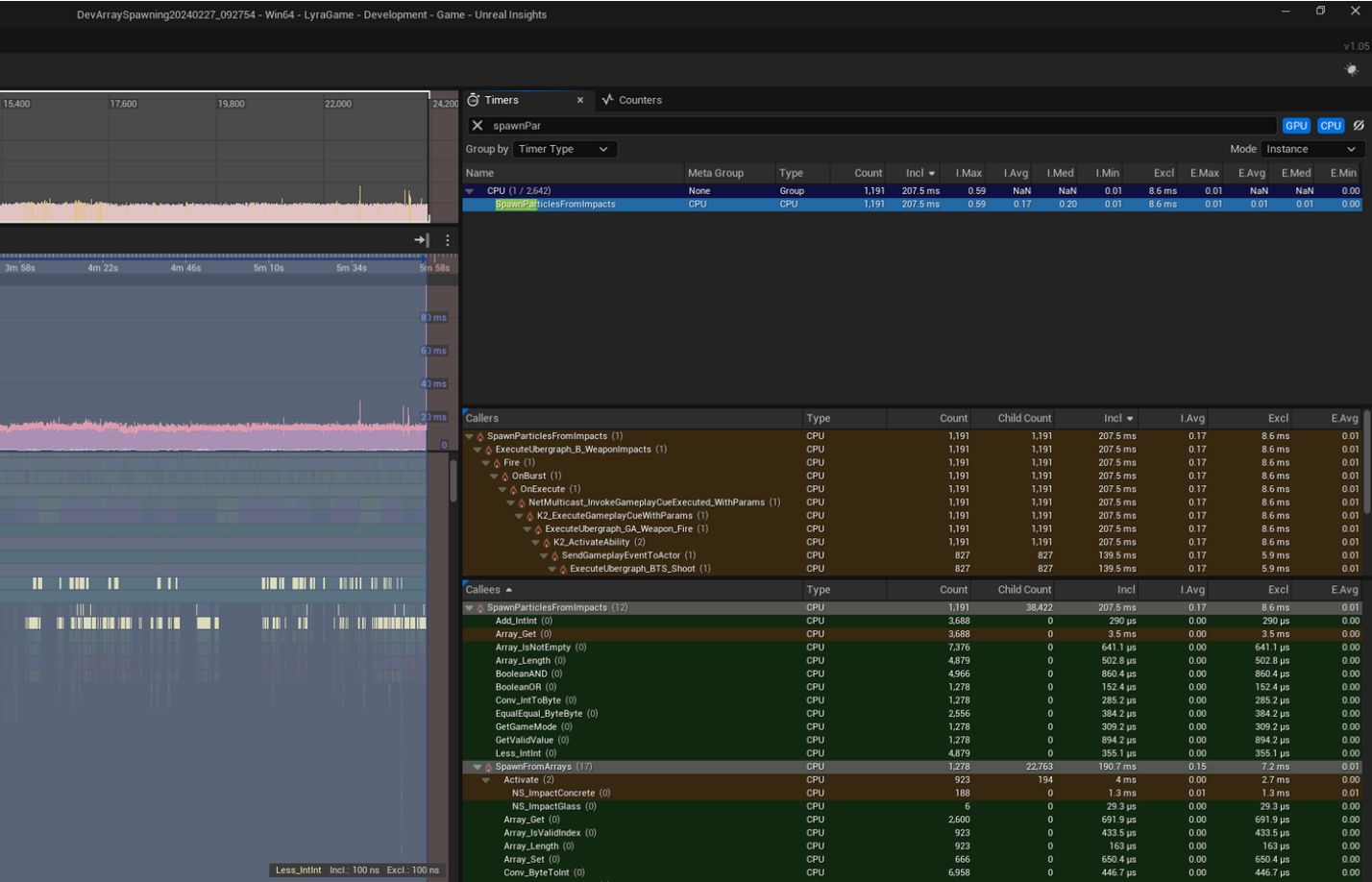
In NS_ImpactDataChannel the module NM_ImpactSpawnDataChannel takes a Min and Max particle count and spawns the burst regardless of the surface, since the behavior is largely the same for both surfaces. NE_ImpactSparksConcrete on the other hand, only spawns from concrete impacts, so it uses NM_SpawnDataChannelConditional to specify a surface type that it should spawn on. Two or more instances of NM_SpawnDataChannelConditional could be used in conjunction to filter the particles to multiple surfaces, or a custom module could be written that has more complex behavior.

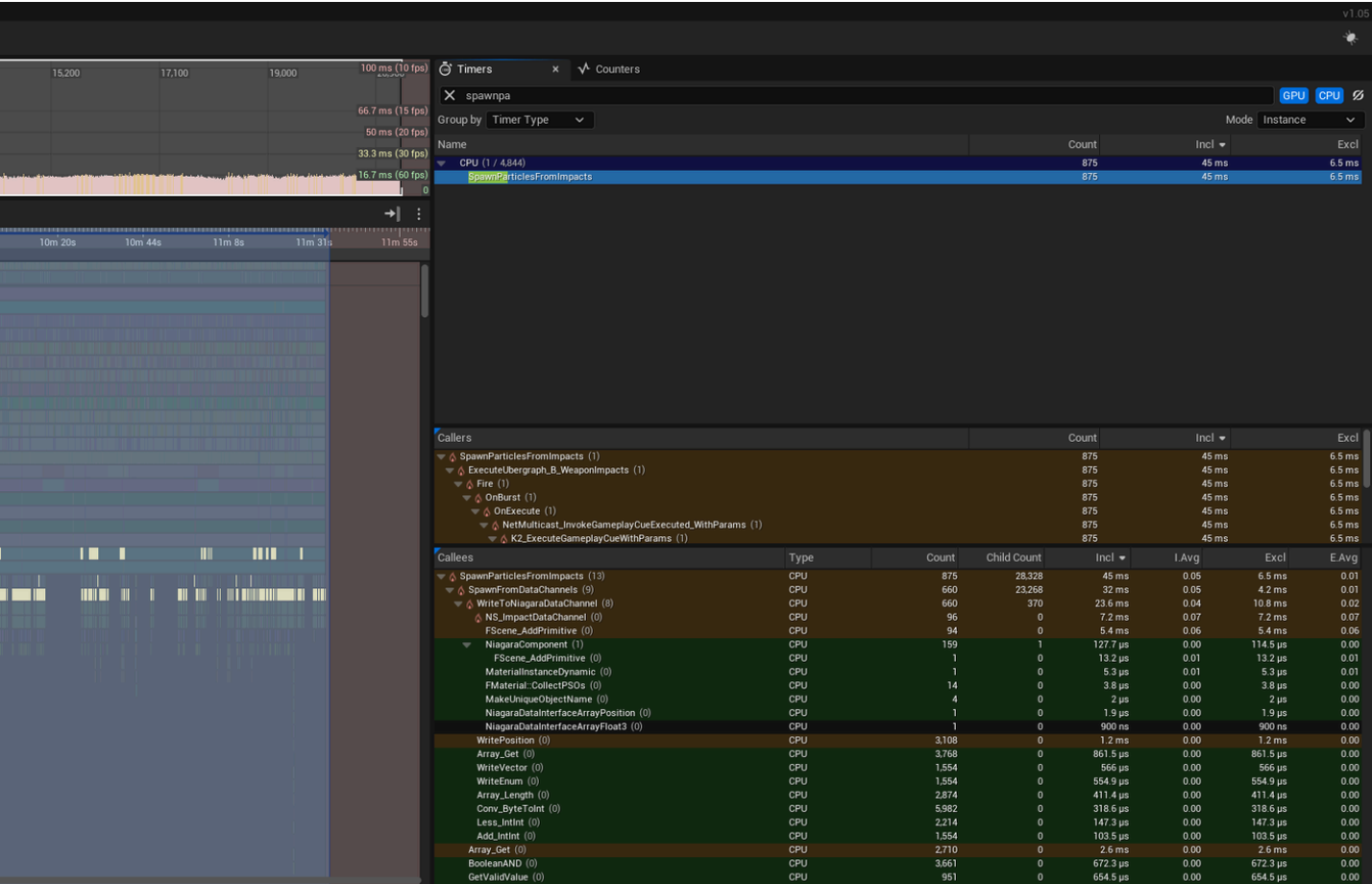
Reading from Data Channels

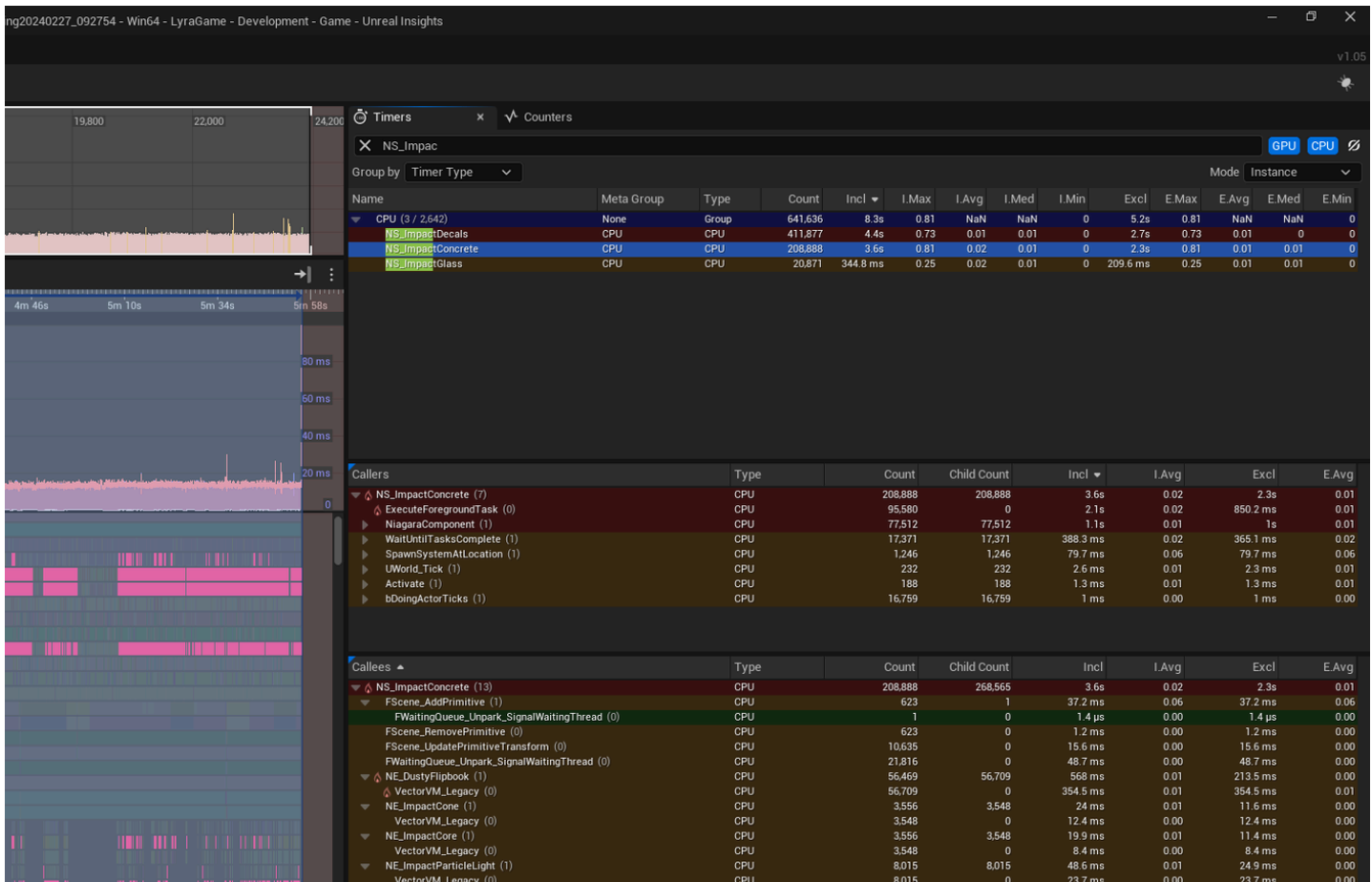
For impacts, the Data Channel data is meant to initialize particles, and is read once in the Particle Spawn script stage with the NM_ImpactReadDataChannel module. NM_ImpactReadDataChannel recreates NM_ImpactSpawnAttributes, just with a different source for its data. It uses the Data Channel Interface stored in the spawn module to read the data using the spawn group as the index. When the particles are spawned with Spawn Conditional it sets their spawn group to correspond to the index of their data in the Data Channel. When reading the data back, it returns a bool if the read was successful, so the module selects from default data, or the data read from the Data Channel based on this.

Instead of using the spawn index, we have now added Get NDCSpawn Data on the Data Channel Reader data interface, that lets you retrieve the Data Channel index based on the Emitter and Particle's exec index when it spawned.

Spawning Performance

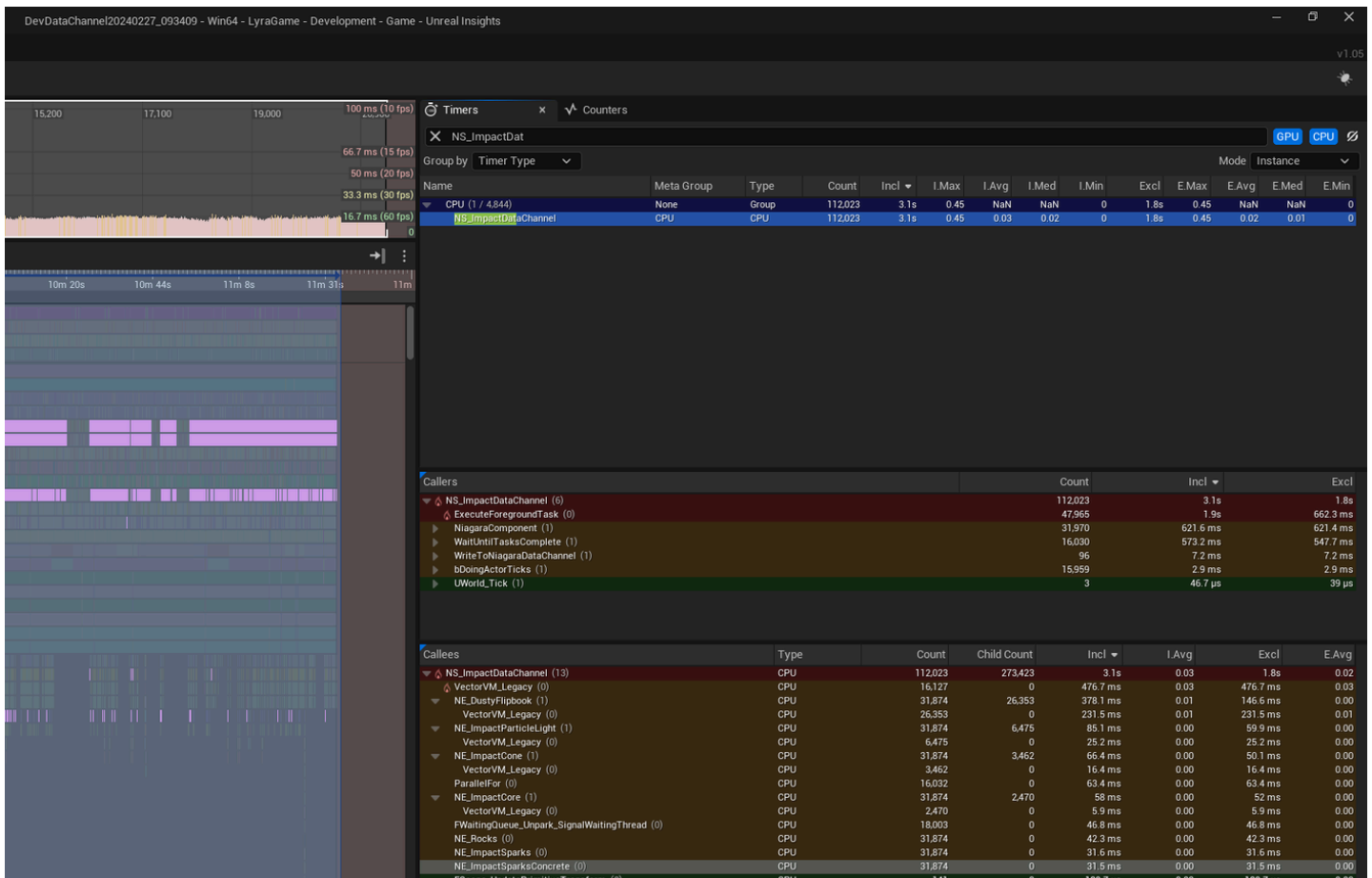






that of glass. This could indicate an issue for merging the systems for data channels, if the overhead for merging glass impacts isn't outweighed by the speed up from reducing system instances, since there aren't many to reduce to begin with, but we pay the overhead with every impact.

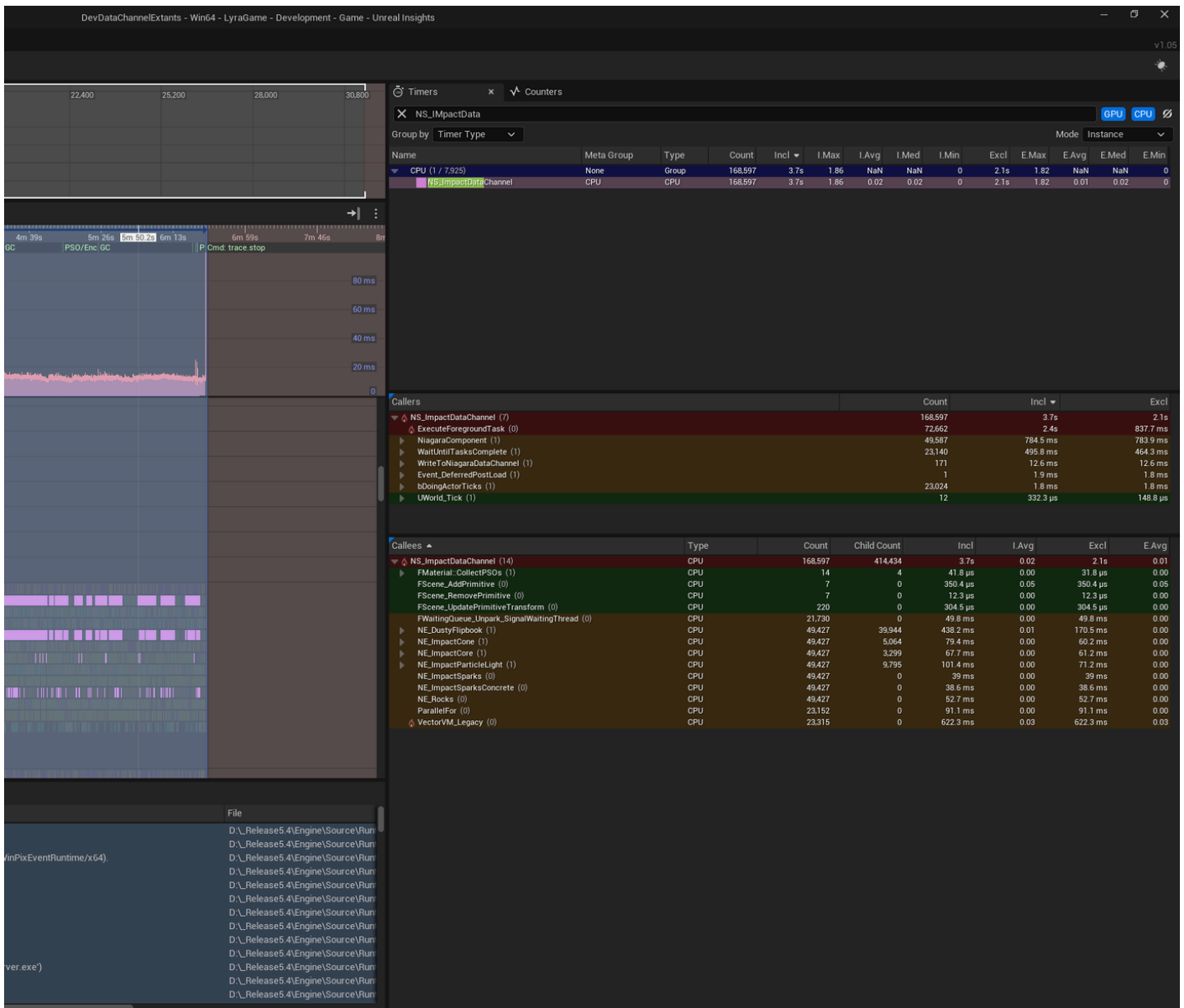
It's worth noting that despite contributing less time on average per frame, the Niagara systems are run far more frequently than their spawning logic, so optimizations here could potentially have a greater impact on the average performance of the game. That being said reducing spikes in frame timing to something more consistent, and amortizing those costs over more frames will lead to more steady frame rates, so it may still be preferable to save on spawning costs at the expense of slightly heavier simulation costs.



systems used in the two approaches have vastly different characteristics.

For array spawning the distance threshold was 500 units, which caused more instances covering less volume each. This is a good case for culling unnecessary particles, but a bad case for instance counts. On the other hand for data channels, the extents were 5000 units on each axis, which lead to fewer instances covering more volume each, which has the inverse result of favoring culling less, and having fewer instances. It's possible, likely even, that some of the per-frame overhead for the NS_ImpactDataChannel comes from running the simulation of many particles that are off screen.

For testing, I halved the extent values to 2500 for the data channels, causing the volume the cover to shrink by 8, and did another test.



In this test their average frame time was .02ms, bringing it back in line with the averages seen in the array systems, and it's spawning characteristics didn't change. This seems to confirm that the original extents were too big, and couldn't leverage culling from scalability as well. There are likely further optimizations that can be done at the asset level for the system, and further tweaks could be made to the extents to optimize these times, but they are low enough already that further time spent is better focused elsewhere. This does demonstrate that consideration still needs to be taken when choosing the correct settings for data channels in your level.