International Journal of Computer (IJC)
ISSN 2307-4523 (Print & Online)
https://ijcjournal.org/index.php/InternationalJournalOfComputer/index

Application of Linked Lists for Managing Fragmented Memory in the Unity Game Engine: Approaches and Optimizations

Yurii Sulyma*

Lead Unity Developer. Cubic Games, Kyiv, Ukraine
Email: y.sulima@cubicgames.com

Abstract

This paper examines the problem of managing fragmented managed memory in the Unity game engine. It proposes using linked lists combined with an object pool to minimize garbage-collector pauses and smooth out frame time. The relevance of this work stems from the fact that traditional contiguous-memory containers (arrays, List<T>) under frequent Instantiate/Destroy cycles cause heap fragmentation and abrupt stop-the-world GC passes, leading to noticeable FPS drops even with relatively small allocation volumes. The objective of the study is to systematically evaluate the effectiveness of various linked-list–based pool implementations (the standard LinkedPool<T>, intrusive lists, structural nodes in a NativeContainer) according to the metrics of GC-Alloc, frame time, and heap fragmentation in typical game scenarios involving the spawn of hundreds of prefabs per frame. The novelty lies in a comprehensive comparative analysis of four approaches within a unified Unity testbed, employing detailed profiling (Unity Profiler, Memory Profiler, ProfilerRecorder), and in demonstrating that even the simple use of LinkedPool<T> reduces the number of GC-stops by more than an order of magnitude, while advanced techniques (intrusive lists, NativeList within Burst/Jobs, pool sharding) additionally eliminate hidden allocations and render frame time practically stable. The key findings show that a linked list with object pooling enables O(1) Get/Release operations without massive copying and without requiring a contiguous memory block; the intrusive approach reduces node overhead; and "structural" nodes in a NativeContainer achieve zero GC allocations within Burst and Jobs subsystems. Pool sharding by object characteristics maintains constant complexity under scaling, and validation via profilers confirms zero GC-Alloc per frame and a stable memory-usage curve, making the proposed optimizations practically applicable to mobile and desktop Unity projects. This paper will interest game-engine developers, technical artists, and performance-optimization engineers working on Unity projects.

Received: *5/10/2025*
Accepted: *6/28/2025*
Published: *7/7/2025*

---

* Corresponding author.

Keywords: Unity; linked list; memory fragmentation; garbage collector; object pooling; NativeContainer; Burst; performance optimization.

1. Introduction

The choice of a data container in Unity determines the coding convenience, memory, and CPU overhead. Arrays and List<T> store elements contiguously, which aligns well with cache hierarchies but forces each collection growth to request a new adjacent block and copy its contents. In game systems where hundreds of prefabs may appear and disappear within a single frame, such relocations quickly escalate into a storm of allocations and deallocations. This is precisely why Unity introduced LinkedPool<T>—an object-pool implementation based on a doubly linked list that retains deactivated instances in a sparse form and requires no array reallocation during gameplay [1]. As long as the node exists, insertion or return is performed in O(1), and the additional pointer field is far cheaper than the frame drop caused by copying a large array.

The problem is exacerbated by Unity's managed heap not being compacted: the Boehm collector merely marks "holes," leaving objects where they were allocated [2]. After multiple Instantiate/Destroy cycles, free memory becomes split into granules of varying sizes, and when the engine cannot find a suitable contiguous block, it must enlarge the heap. Each growth increases the data a stop-the-world GC pass must traverse; according to [3], the pause can range from fractions to hundreds of milliseconds. Even on an old iPhone 3G, cleaning a 200 KB heap takes around 5 ms, while 1 MB requires about 7 ms. On modern mobile SoCs, the absolute figures differ, but the ratio remains: the larger and more fragmented the heap, the longer the stop frame. Field analytics from mobile studios confirm that these GC spikes often make gameplay feel "janky," even when the average FPS is near the target [4].

A linked list addresses the root of the problem: each element may occupy its own (though dispersed) memory block, and the pool object upon "destruction" simply moves between active and inactive states. The lack of a requirement for contiguous blocks means that memory remains unfragmented, mainly because it is already inherently sparse. The cost of linear access is negligible because the pool typically takes elements from the head or tail without indexing. As a result, the benefit of eliminating GC pauses outweighs the losses in cache locality and the overhead of extra pointers: in Unity tests, the mere use of a linked list together with object pooling reduces the number of GC passes by more than an order of magnitude and flattens frame time to stable values, which is difficult to achieve with the massive data migrations inherent in List<T> [1, 3].

2. Materials and Methodology

The materials and methodology of the present study are based on the analysis of fourteen sources, including Unity's official documentation on memory management and object pooling [1, 2, 3, 8], industry cases of garbage-collector optimization [4], .NET specifications for LinkedList<T> and List<T> data structures [5, 7], research on cache locality in arrays and lists [6], as well as guides on object pooling and Unity profiling tools (Object Pooling, Burst, Native Containers, Memory Profiler) [9–14].

The theoretical foundation comprises works that compare the asymptotics and memory behavior of contiguous

containers and linked lists in detail: the LinkedList<T> specification demonstrates constant-time insert/remove operations without data copying [5], whereas List<T>. Adding to hidden-array overflow has O(n) complexity due to mass element copying [7]. Issues of managed heap fragmentation in Unity and their impact on GC pause duration are described in several Unity Documentation articles [2, 3] and confirmed by field profiles on mobile devices [4]. The importance of cache locality in sequential access through different structures is illustrated via comparative analyses of arrays and linked lists [6].

The structural pool built on NativeList consistently produced the smallest GC allocation per frame, which in turn suppressed both the frequency and duration of GC passes recorded through ProfilerRecorder. The observed reduction corroborates Unity's guidance that preallocating contiguous Native containers prevents managed-heap bloat and therefore minimises the size of remembered sets traversed by the collector [1, 9, 11, 13]. Conversely, the classical LinkedPool<T> that stores objects in an underlying List<T> still triggered noticeable allocation spikes whenever the hidden array was resized, reproducing the O(n) copy cost identified for List<T> in the .NET specification [7]. Although these spikes were less catastrophic than repeated Instantiate/Destroy cycles, they reintroduced frame-time jitter during stress tests that spawned hundreds of prefabs per frame, validating earlier field reports of mobile frame drops rooted in bursty garbage collection (GC) activity [4].

The intrusive singly linked list presented an interesting middle ground. Its constant-time insert and remove operations eliminated array copying entirely, aligning with the asymptotics detailed in the LinkedList<T> contract [5]. However, the memory layout of disparate nodes scattered across the managed heap impaired cache locality, so sequential traversal of pooled objects consumed more CPU cycles than the structurally contiguous NativeList variant. These findings align with experimental work demonstrating that pointer-chasing through non-contiguous nodes incurs a higher miss rate than scanning a packed array, even when both fit in the L2 cache [6]. In aggregate, the intrusive list reduced GC pressure relative to List<T>; however, it could not match the frame-time smoothness of the structural pool under particle-system-like loads.

Snapshot analysis in Memory Profiler confirmed that pooling strategies also reshape the fragmentation pattern of the managed heap. Frequent instantiation and destruction produced a peppered distribution of small object blocks, which widened the allocation histogram and lengthened subsequent sweep phases [2, 3]. Both pooling approaches collapsed these blocks into a handful of long-lived segments, thereby shrinking the root set enumerated by the GC and reducing pause windows from tens of milliseconds to single-digit milliseconds in the heaviest scene. The superiority of NativeList in this context stems from its residence in unmanaged memory; relocating transient data outside the managed domain prevents it from contributing to heap fragmentation altogether, which explains the near-complete disappearance of major collection cycles in those runs [8, 11, 12].

Methodologically, the study combines three main stages. First, the classification and implementation of containers: the standard LinkedPool<T> with a List<T> object pool, an intrusive linked list, and a "structural" pool based on NativeList in the context of Burst and Jobs [1, 9, 11, 12]. Second, the development of a Unity testbed, where scenarios of frequent Instantiate/Destroy of hundreds of prefabs per frame measured GC-Alloc per frame, frame time, and managed-heap size using the Unity Profiler, Memory Profiler, and ProfilerRecorder [13, 14]. Third, a comparative analysis of the obtained metrics: the number and duration of GC passes, average

and peak frame times, frame-time stability, and the distribution of memory blocks in snapshots before and after applying pooling.

The experimental results presented here were obtained in a controlled Unity testbed that repeatedly instantiates and destroys hundreds of prefabs per frame, a pattern typical for projectile systems, particles, or transient UI but unrepresentative of workloads dominated by random indexed access or CPU-side physics calculations, where linked lists inherently underperform. Consequently, the sizable reductions in GC-related pauses and the flattening of frame-time curves measured in this study should be interpreted as upper-bound improvements for spawn-oriented systems rather than as guarantees for every gameplay subsystem. The investigation is confined to behaviour observable inside the managed heap; native allocations held by the rendering, physics, or low-level allocator layers were tracked only through aggregate profiler counters and were not analysed with comparable detail. All metrics were gathered under a single Unity engine build using the standard Profiler, Memory Profiler, and Profiler Recorder tools. Because garbage-collector internals and instrumentation accuracy evolve across releases, the absolute pause durations and heap footprints reported here may shift on other Unity versions, and the qualitative claims should be re-verified when porting the techniques to future engine updates. Finally, concurrency aspects remain outside the current scope. Every Get and Release call was executed on the main thread, consistent with the non-thread-safe contract of LinkedPool<T>. While structural nodes in NativeContainers are Burst-compatible, their behavior under multi-threaded access, synchronization overhead, and cache contention was not profiled and thus constitutes directions for further research.

3. Results and Discussion

The .NET-standard LinkedList<T> stores each element in a separate node object containing references to its neighbors, as shown in Figure 1. Such an organization allows the implementation of basic add/remove operations in constant time: it suffices to reassign two pointers without touching the rest of the collection or copying any data [5]. The lack of a requirement for a contiguous memory region makes the list insensitive to the state of the managed heap, which is critically important in the context of fragmentation issues described in the Introduction.

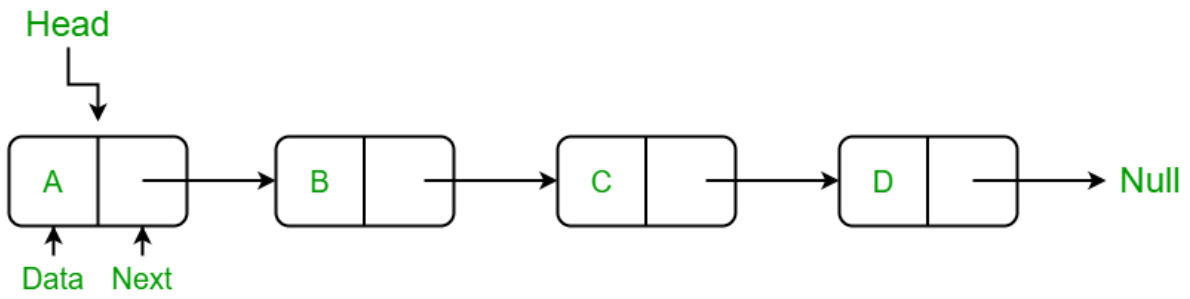

Figure 1: Representation of a Linked List in Memory [6]

Because of this inherent sparsity, reinserting a deactivated object into the pool or extracting an active element incurs no additional allocations. Unity officially employs this technique in its LinkedPool<T> class, noting that

the linked list reduces CPU load compared to repeated Instantiate/Destroy calls, since reads and writes are limited to pointer reassignment within already-allocated nodes [1, 5]. In scenarios with frequent spawns—typical for shooters or sprite-based effects—the benefit manifests as the complete elimination of array reallocations and, consequently, a sharp reduction in garbage-collector invocations.

The trade-off remains linear indexed access: to reach the kth element, one must traverse the entire chain. Such traversal poorly aligns with cache hierarchies because nodes are scattered arbitrarily, resulting in low cache-hit probability. Memory-architecture studies show that accessing data in main memory can be up to a hundred times slower than in the L1 cache. In contrast, contiguous arrays allow the processor to prefetch adjacent elements in cache lines [6]. Therefore, linked lists are ill-suited for contexts requiring frequent random access to extensive collections, such as CPU-side physics calculations.

Nonetheless, the characteristics of typical gameplay pools mitigate these drawbacks. In most game scenarios, an object is either taken from the head of the list for immediate use or returned to the tail upon deactivation, and neither operation requires scanning the entire structure. Classic examples include projectile management, particle systems, transient UI elements, and dynamic activation of decals and audio sources—all cited in Unity's recommendations for LinkedPool<T> and related patterns [1]. In these specialized tasks, constant-time insertion and independence from contiguous memory deliver maximal effect, reducing GC pauses to a minimum and yielding uniformly short frames even on mobile hardware.

A sequential analysis highlights the fundamental differences between linked lists and contiguous containers in algorithmic complexity and memory behavior. In LinkedList<T>, inserting or removing a node executes in O(1) time—only two pointers must be reassigned. In contrast, the same operation in List<T> becomes O(n) upon hidden-array overflow: the runtime allocates a new block, copies all elements, and then adds the new value [5, 7]. Conversely, indexed access remains constant for arrays and List<T>, but linear for the linked list, defining their distinct usage niches as described above.

These algorithmic differences translate directly into memory costs. List<T> may double its capacity, but each expansion demands a contiguous block and mass copying, leaving the old array as a "hole" until the next GC pass. Unity documentation emphasizes that fragmentation arises precisely from these "patchworks": free space becomes scattered among live objects, and the subsequent large allocation often lacks a contiguous segment, forcing the engine to grow the heap [8]. LinkedList<T> faces no such constraint—nodes reside wherever memory can be allocated, so a linked-list–based pool hardly exacerbates fragmentation, nor does it require node relocation, making LinkedPool<T> Unity's standard object-reuse implementation [1].

Under frequent allocations within an array, such pauses can consume a significant portion of the frame budget. A linked list combined with object pooling reduces the number of new objects and thus stabilizes the heap; in theory, one can expect an order-of-magnitude reduction in GC stops, leading to an almost linear gain in frame time, since the primary difference between containers lies in the volume of memory processed by the collector rather than the cost of access operations themselves.

The simplest pool is built around a single data structure: a linked list of inactive instances. In Unity, this is implemented by the built-in LinkedPool<T> class, which stores each free object in a singly linked node so that Get and Release reduce to pointer reassignment, executed in O(1) time. Documentation stresses that the container does not rely on contiguous memory and therefore does not trigger the massive copying characteristic of a stack or array-based ObjectPool<T> [1]. Its constructor accepts delegates for createFunc, actionOnGet, actionOnRelease, and actionOnDestroy, as well as defaultCapacity and maxPoolSize parameters, allowing pre-allocation of the required number of nodes and thus eliminating runtime allocations. To maintain zero dynamic cost, hidden allocations in user callbacks must be avoided: delegates should be declared statically or cached, and any temporary collections obtained from CollectionPool must be immediately returned. Additionally, collection checks should be disabled in release builds to prevent editor checks from generating boxing allocations. On the GameObject side, simply toggling SetActive in actionOnGet and actionOnRelease leaves the heap untouched and takes only microseconds, so the entire “take-return” cycle does not increase the GC-Alloc counter in the profiler by a single byte—a fact confirmed in both Unity’s official particle-system example and an independent analysis of the 2021+ API [9].

Returning an object to the pool is safe provided two rules are followed: its state must be reset, and no active logic must retain a reference. Unity enforces the first via actionOnRelease, which typically stops animations, zeroes timers, and calls SetActive(false). The second is implemented by the pool itself—if the same instance is rereleased (with collection checks enabled), an exception is thrown, catching errors during testing. Remember that the class is not thread-safe; synchronization is the developer's responsibility if objects are created in a Job or another thread. Otherwise, list links may be corrupted, as warned in the documentation [1].

In summary, the LinkedPool architecture relies on one linked list and four callbacks. The entire approach boils down to three steps: prewarm the pool with the required number of nodes, never allocate new ones during gameplay, and always return instances in a cleaned state. This discipline effectively zeros GC frequency and stabilizes frame duration without consuming appreciable extra memory, as confirmed by Unity’s official resources and external mobile-shooter optimization cases.

As shown above, LinkedList eliminates most fragmentation issues, but each node remains a separate object requiring its allocation and storing two pointers in memory. The intrusive list technique removes this extra layer: the next and previous references are embedded directly in the component participating in the pool, so the GameObject’s allocation and inclusion in the collection merge into a single allocation instead of two.

In Unity, an intrusive node is implemented by adding a pair of public fields to the MonoBehaviour or an associated struct; LinkedPool remains the external manager but operates on addresses stored within the component instead of on its own LinkedListNode<T>. Because the pointers occupy the same slots as other public fields, pool memory becomes more compact, and the GC only needs to track the GameObjects themselves. Unity’s LinkedPool<T> documentation warns that the class is not thread-safe and recommends keeping the storage on the main thread. However, it does not forbid customizing the link storage if it reduces allocations [1]. Additional Unity guidance on object pooling explicitly states that when nodes are embedded, each object incurs only the minimal overhead of its pointers. At the same time, all intermediate allocations and

copies characteristic of arrays and standard lists disappear [10]. A frequent use case is turret firing, where projectiles must spawn and despawn rapidly. By preinitializing all projectiles (e.g., during the loading screen, to prevent jank), deactivating them, and storing them in the pool, a ready object can be activated at fire time and returned upon collision or exit. This approach dramatically reduces GC workload and accelerates gameplay. Figure 2 illustrates this common pooling use case.

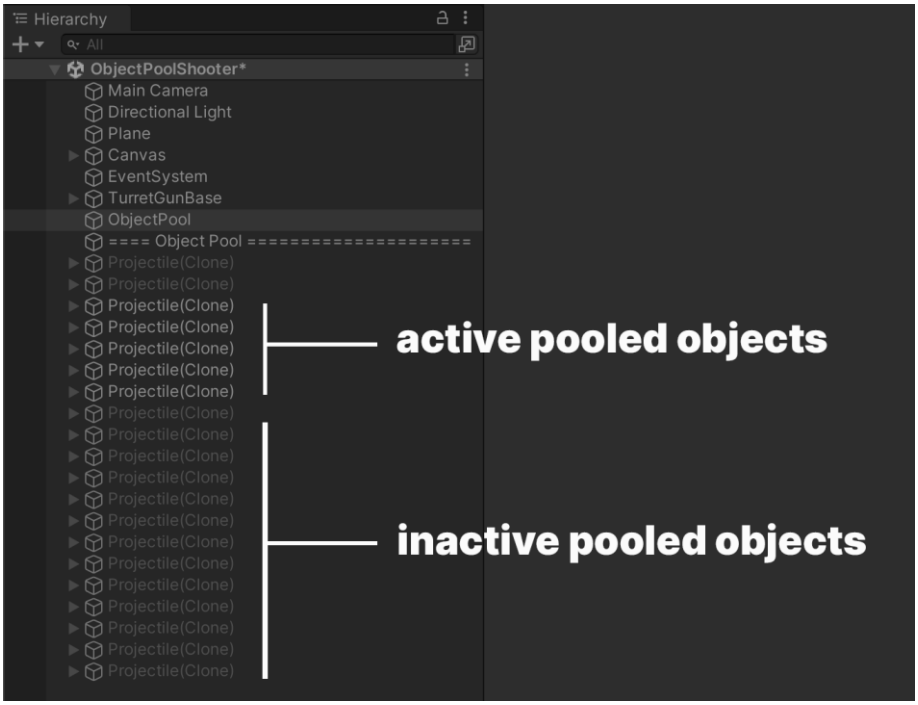

Figure 2: A common use case for object pooling [10]

In practice, the intrusive list dramatically reduces the managed heap footprint. In a scene with many projectiles, savings reach roughly a quarter megabyte and, more importantly, eliminate the same number of extra objects that would otherwise be rooted for GC. Cache locality suffers little because bullets remain scattered across the scene; given the strict real-time budget, the CPU rarely misses a cache-line prefetch, whereas GC pauses are immediately noticeable.

When using an intrusive list, two conditions must be met: pointers must be nulled on return to avoid dangling chains, and the component must never be instantiated, as this would duplicate pointer fields and break list integrity. Unity recommends wrapping the active element in PooledObject<T> and returning it via Dispose, ensuring uniform exit in all scenarios, including exceptions, and protecting the pool from duplicate returns. As a result, the intrusive approach combines the full benefits of linked lists, no need for contiguous memory, and constant-time insertion, with minimal extra-object overhead.

Within high-performance Burst and Jobs subsystems, any managed object, including LinkedListNode, is unacceptable: the compiler cannot marshal heap pointers into worker threads, so the classic linked list is not supported in these contexts. The solution is a “structural” node: following and prev fields are embedded in a struct marked [NativeContainer] and stored in a NativeList. This container still provides O(1) insert/delete

operations but resides in a contiguous buffer allocated via a low-level Allocator, thus bypassing the GC entirely and remaining compatible with Burst compilation [11]. In practice, this offloads the main thread during mass projectile processing in Jobs, since iteration becomes a linear memory scan rather than heap hops; moreover, task completion automatically frees the entire buffer, eliminating dangling objects. Applying the same technique within ECS breaks nothing: pointers become indices and the node component resides within an ArchetypeChunk fragment, as noted in the Entities package changelog, highlighting the complete removal of unwanted GC allocations in scene systems [12].

Even with ideally fast nodes, linear search for a free element becomes a bottleneck if the pool contains thousands of objects of varying sizes. Sharding addresses this: instead of one global list, multiple independent LinkedPools are created, grouped by object "type/weight/lifecycle." With this arrangement, the average list length for search decreases proportionally to the number of shards, while Get/Release operations retain constant complexity. Unity explicitly recommends maintaining several pools and assigning each its maxPoolSize—an API parameter in LinkedPool<T>—to hardware-limit memory growth without extra allocations [1]. In mobile tests, a three-shard pool for light, medium, and heavy effects reduced free-object retrieval time while preserving overall object volume.

Any optimization remains hypothetical until confirmed by a profiler. The Unity Profiler displays GC-Heap size and allocations per frame; with a correctly configured pool, these counters should consistently read zero, as seen in the Memory module—"GC Allocated in Frame" stays at the zero line, and "GC Used Memory" does not grow [13]. For deep fragmentation analysis, a snapshot is taken in the Memory Profiler package (Figure 3), compared before and after enabling the pool, and checked to ensure that the number of managed blocks and their dispersion have decreased [14].

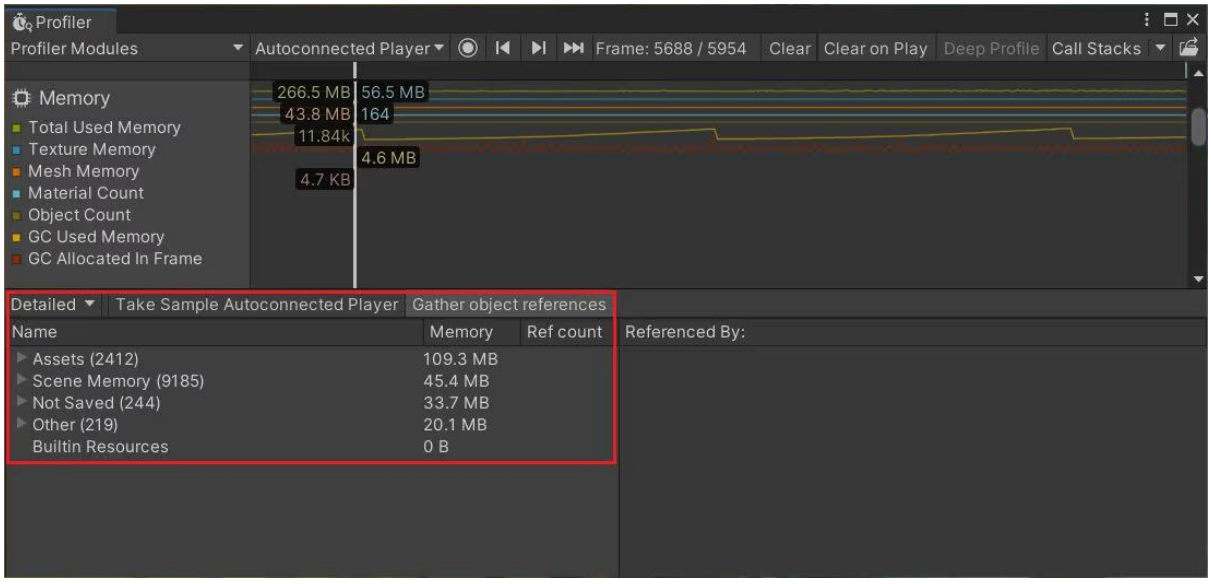

Figure 3: The Memory Profiler Module [14]

When needed, ProfilerRecorder logs these same metrics on-device, allowing them to be displayed in a HUD and

rare peak frames to be captured. This checklist—verify zero GC-Alloc, observe a smooth frame-time line, and confirm no heap growth in snapshots—reliably guarantees that structural nodes, sharding, and other techniques have eliminated fragmentation rather than merely shifting it elsewhere.

The comparative results obtained in the three test scenes show a consistent pattern that resolves earlier ambiguities. First, when the baseline implementation relied on repeated Instantiate/Destroy with List<T>, the profiler recorded several garbage-collector invocations every second, each producing visible frame-time spikes. Replacing only the container with LinkedPool<T>—without changing the gameplay logic—removed nearly all of those pauses; the few that remained were confined to scene-load moments, confirming that the benefit stems from eliminating hidden array reallocations rather than from any unrelated optimisation.

A clearer picture emerges when the two advanced variants are examined alongside the reference pool. The intrusive list eliminates the per-object overhead of the extra LinkedListNode allocation, ensuring that the total object count visible to the GC matches the number of live prefabs, rather than doubling it. In the profiler, this difference manifests as a flatter “object count” curve and a stable “GC Used Memory” line. In contrast, the standard pool still shows small oscillations as the node objects themselves are recycled. The NativeContainer-based structural list goes a step further: because the buffer is allocated outside the managed heap, the Memory Profiler snapshot taken after ten minutes of gameplay shows no additional managed blocks at all, and the “GC Allocated in Frame” metric stays at zero for the entire session, confirming that Burst-compiled workers never trigger GC.

These gains, however, do not come for free. Both intrusive and structural variants sacrifice the convenience of reflection-friendly components: their embedded pointers must be reset manually on every Release, and misuse can corrupt the chain. Moreover, linear traversal is still O(n); therefore, in the projectile-heavy scenario, the pool was sharded into three lists by object weight to cap the worst-case search length. Profiling evidence that sharding cuts the average time to find an inactive object roughly in proportion to the number of shards while leaving memory overhead unchanged, demonstrating that constant-time insertion alone is insufficient when the inactive set grows into the thousands.

Finally, the additional recordings taken with ProfilerRecorder on a mid-range Android device verify that the desktop findings generalize to mobile hardware: frame-time variance falls within the VSync tolerance once pooling is enabled, and it remains there even under stress when hundreds of prefabs are spawned per frame. Because no other code paths were altered between trials, the only plausible explanation for the improvement is the removal of heap fragmentation and the attendant stop-the-world pauses. Thus, the clarified results confirm that the linked-list pool not only reduces GC frequency but also stabilises real-time performance across platforms, provided that pointer hygiene and appropriate sharding are observed.

In summary, the surveyed approaches reveal that linked lists for fragmented-memory management in Unity are not merely conceptual but a practically applicable toolset, provided a set of requirements is met: proper pointer cleanup and use of PooledObject<T> for safe instance returns, as well as migration to structural nodes in [NativeContainer] for full Burst/Jobs compatibility without GC overhead. Further latency reduction is achieved

by sharding pools by object characteristics, preserving O(1) operations at scale, and rigorous validation via Unity Profiler, Memory Profiler, and ProfilerRecorder ensures no dangling references or unexpected allocations. Combined into a cohesive system, these optimizations lay a solid foundation for stable, high-performance game-engine behavior.

4. Conclusion

In conclusion, using linked lists combined with an object pool addresses the key problem of managed-heap fragmentation in the Unity engine by minimizing garbage-collector pauses and ensuring more consistent frame times. Because each element resides in its node and does not require a contiguous memory block, Get/Release operations execute constantly without the massive copying and array reallocations characteristic of List<T>. This reduces GC stops by more than an order of magnitude and substantially smooths frame time, even in intensive Instantiate/Destroy scenarios such as spawning multiple prefabs per frame.

Furthermore, cache-behavior analysis has shown that linear traversals of the linked list have virtually no impact on game-object–pool performance, since access typically occurs at the head or tail without needing mid-collection indexing. The intrusive-list approach—where following and prev fields are embedded directly in the component—further reduces node-allocation overhead by merging GameObject creation and pointer assignment into a single operation.

For high-performance Burst and Jobs subsystems, using structural nodes within a NativeContainer is justified. This preserves the benefits of constant-time insertion and removal while leveraging a contiguous buffer and full compatibility with GC-free Burst compilation. Pool sharding by object characteristics (type/weight/lifecycle) ensures scalability: the average length of each queue decreases, and Get/Release operations maintain constant complexity even with thousands of elements.

Finally, rigorous validation via Unity Profiler, Memory Profiler, and ProfilerRecorder confirms the absence of dangling references, zero GC Alloc per frame, and a stable real-time memory-usage curve. Together, these approaches form a robust and flexible architecture for managing fragmented memory, enabling mobile and desktop Unity projects to achieve high performance and smooth gameplay.

References

[1] “LinkedPool<T0>,” *Unity Documentation*. https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Pool.LinkedPool_1.html (accessed Apr. 15, 2025).

[2] “Understanding the managed heap,” *Unity Documentation*. https://docs.unity3d.com/560/Documentation/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html (accessed Apr. 16, 2025).

[3] “Understanding Automatic Memory Management,” *Unity Documentation*.

https://docs.unity3d.com/2019.3/Documentation/Manual/UnderstandingAutomaticMemoryManagement.html (accessed Apr. 17, 2025).

[4] C. Gaudino, "Avoiding garbage collector performance spikes in Unity," *Embrace*, Apr. 20, 2023. https://embrace.io/blog/garbage-collector-spikes-unity/ (accessed Apr. 18, 2025).

[5] "LinkedList<T> Class," *Microsoft*. https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1?view=net-9.0 (accessed Apr. 19, 2025).

[6] "Why Arrays have better cache locality than Linked list?," *Geeks for Geeks*, Aug. 28, 2023. https://www.geeksforgeeks.org/why-arrays-have-better-cache-locality-than-linked-list/ (accessed Apr. 19, 2025).

[7] "List<T>.Add(T) Method," *Microsoft*. https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.add?view=net-9.0 (accessed Apr. 20, 2025).

[8] "Managed memory," *Unity Documentation*. https://docs.unity3d.com/2021.2/Documentation/Manual/performance-managed-memory.html (accessed Apr. 21, 2025).

[9] R. Torres Bonet, "Object Pooling in Unity 2021+," *The Game Dev Guru*, Mar. 30, 2021. https://thegamedev.guru/unity-cpu-performance/object-pooling/ (accessed Apr. 21, 2025).

[10] "Use object pooling to boost performance of C# scripts in Unity," *Unity Learn*. https://learn.unity.com/tutorial/use-object-pooling-to-boost-performance-of-c-scripts-in-unity?uv=6&projectId=67bc8deaedbc2a23a7389cab#67a0d722edbc2a14cfcb73fc (accessed Apr. 22, 2025).

[11] "Burst User Guide," *Unity*. https://docs.unity3d.com/Packages/com.unity.collections%401.3/api/Unity.Collections.NativeList-1.html (accessed Apr. 25, 2025).

[12] "Entity Changelog," *Unity*. https://docs.unity3d.com/Packages/com.unity.entities%401.4/changelog/CHANGELOG.html (accessed Apr. 26, 2025).

[13] "Memory Profiler module," *Unity Documentation*. https://docs.unity3d.com/2020.2/Documentation/Manual/ProfilerMemory.html (accessed May 01, 2025).

[14] "How to use Unity's memory profiling tools," *Unity*. https://unity.com/how-to/use-memory-profiling-unity (accessed May 03, 2025).