Performance checklist

Things to consider when your code is slow.

Have you profiled? Where are the bottlenecks?

The first rule of performance is always:

> *Don't guess! Profile!*

Is your code slow? How slow? Where is it slow? Until you measure the actual performance of your code on representative hardware with profiling tools, you should not trust your own speculative answers to these questions. Sure, intuition and experience can help guide your investigations and optimization efforts, but ultimately, you should verify everything with actual profiles.

Failing to profile may lead to wasting time and energy on ineffective optimizations, possibly even on the entirely wrong parts of code. For example, optimizing CPU code probably won't help your overall performance much if your game is GPU-bound, or *vice versa*.

Have you budgeted for performance?

When profiling a game with unsatisfactory performance, you may sometimes find that there are no clear, standout bottlenecks in the code: the slowness is spread evenly throughout. This is unfortunate, because it implies that fixing performance will require either:

- Scaling back features or fidelity.
- Rewriting every separate part of code to avoid inefficient coding practices.
- Rewriting the code as a whole to correct foundational architectural mistakes.

To avoid getting into this painful situation, you should budget and measure performance starting from the earliest stages of development. 'How many milliseconds of CPU can a particular feature be allowed to take in a single frame? How much memory should the feature require?' *Etc.* These are questions you should try to answer as soon as possible.

Are potentially expensive things done one-at-a-time?

Conventional OOP programming patterns tend to encourage processing elements one-at-a-time, and this often hurts performance and thwarts optimization efforts.

For example, consider simulation updates for monster NPCs, where the logic for each monster requires several complex, expensive steps, such as path finding, AI behaviour, collision detection, and checking line-of-sight to the player character. The most obvious, natural way to structure such logic in OOP is with an update method that performs each of these steps for just an individual monster. In Unity terms, this would be a MonoBehaviour that looks something like this:

```
Java
public class Monster: MonoBehaviour
{
	// in a frame, called for each monster
	void Update()
	{
		// each call processes just one monster
		pathFinding();
		aiBehaviour();
		collisionDetection();
		lineOfSight();
	}
}
```

Alternatively, each step of monster logic could process all monsters as a batch:

```
Java
public class MonsterManager: MonoBehaviour
{
	// in a frame, called only once for all monsters
	void Update()
	{
		// each call processes all the monsters
		pathFinding(monsterArray);
		aiBehaviour(monsterArray);
		collisionDetection(monsterArray);
		lineOfSight(monsterArray);
	}
}
```

This ‘batching’ or ‘grouping’ pattern has several potential benefits:

1. The data required for a given task will now generally be loaded from memory into cache fewer times per frame. For example, the navmesh used in pathfinding can be loaded once instead of separately for each monster.
2. The code itself for each step will now likely only be loaded from memory into cache just once per frame.
3. The code of a given task can often handle batches more efficiently than if the same task were repeated separately for each individual. For example, multiple line-of-sight checks to the same single point might be more efficient when batched.
4. Because the steps of monster logic are clearly separated, the bottleneck steps can be more easily identified in profiles.
5. Because the different steps likely use different subsets of the monster-related data, separating the steps likely makes the code easier to multi-thread.

On the downside, this pattern does mean that you now traverse the monster array multiple times, once for each step, rather than just once. In some cases, then, you might consider recombining some or all of the steps. As a general strategy, however, you’re best off starting with the steps split out, and you should only later combine steps if profiles verify that doing so would actually improve the performance.

Have you prioritized the common cases?

Sometimes, the various features of your program impose conflicting choices: making one feature performant and simple to implement may come at the cost of making another feature less performant and more complicated. Given the choice, then, you should prioritize “common” cases over uncommon cases because the things which your program needs to do frequently and at large scale tend to have the highest potential performance costs.

Designing your key data structures and code around the common cases at the outset of your project makes it much more likely that your solution prioritizes the most important performance problems. Neglecting to prioritize common cases up front can become more and more costly as your project progresses because you may end up baking in bad assumptions about what’s most important that can only be corrected later by heavy rewrites.

Are you paying for unnecessary work?

Are you paying for excessive algorithmic time-complexity?
Are you paying for excessive constant-time overhead?

For some problems, different solutions perform radically different amounts of work and thus likely very different amounts of processing time. For example, whereas a brute force search may require exhaustively looking through every single item, a binary search at worst has to look through only log(n) of the items.

For other problems, however, skipping items of work is not as important as the “constant time” factor, *i.e.* the cost of processing each item. Particularly in games, brute force solutions with low constant time factors may be preferable to more clever solutions.

Can expensive work be done at build time instead of runtime?
Can the result of expensive work be cached and reused later?

Instead of paying for expensive processing at runtime, maybe the processing can be done up front before the program even runs.

Similarly, instead of doing the same expensive processing repeatedly, maybe the results can be saved and reused later in the same frame or in subsequent frames.

Can expensive work be done less frequently?

In some cases, expensive work only needs to be done intermittently, though perhaps at the cost of less accurate results. For example, if NPC pathfinding is updated only once every second instead of once every frame, the resulting behaviour may be less “correct” but perhaps not noticeably so.

Can expensive work be deferred and consolidated?

Related to the idea of batching like-work-with-like, it may be beneficial to separate out expensive cases. For example, when processing monster NPC’s, you may find that some monsters need much more complex processing than others:

```
Java
void doTaxes(monsterArray)
{
	for (var monster : monsterArray)
	{
		// ...process a monster's taxes
```

```
            // If it's determined that the this monster's taxes requires
        complex processing,
            // the work is deferred untill later rather than done here.
            if (...)
            {
                // ... mark the monster or add the monster to a queue to be
            processed later
                continue;
            }

            // ...finish simple case
        }
    }
```

Having split out the hard cases, you can then process them in a follow-up step.

This pattern can pay off in a few ways:

1. Some cases may require radically different logic, so separating them out likely makes the code clearer.
2. Different cases may require access to very different data, *e.g.* simple taxes only require pay stubs, but complex taxes require a mountain of receipts and financial documents. Thus, separating out the cases may help minimize the scope of data access in the respective parts of code.
3. Separate cases may have very different performance characteristics, so separating them makes bottlenecks easier to identify in profiles.

Even if separating out the expensive cases doesn't itself directly improve the performance, making the code clearer with better data isolation will generally make the code easier to optimize.

Are you using memory efficiently?

Because memory access is costly relative to the operations of the CPU, memory usage is one of the most impactful factors of performance. To minimize the costs of memory access, the general strategy is to:

- avoid reading memory
- avoid writing memory

- minimize the size of your data (which helps keep down the total memory footprint and helps avoid reads and writes of memory)

Here are a few more specific things to watch out for:

- In some cases, you may be able to minimize how often a collection is traversed. For example, if a large array is looped through multiple times in a single frame, maybe the data could instead be traversed just once in a single frame.
- Consider the overall size, packing, and memory layout of your data types (structs and classes). In some cases, it helps to shrink a data type by splitting off some of its data into separate additional types.
- Algorithms which minimize the amount of computation work do not necessarily maximize memory efficiency. When choosing algorithms, consider not just the time-complexity and constant time factors but also the amount of space they require and how much they read and write memory.
- When possible, prefer traversing data sequentially rather than jumping around different parts of memory. The “random lookups” of non-sequential traversal will typically trigger many more cache misses.
- If you have many boolean flags, consider packing them into bit fields instead of using booleans.

Are you paying for excess allocations?

Even without garbage collection, the costs of memory allocations can add up, for several reasons:

- When your program needs more memory from the operating system, this requires a system call, which incurs significant overhead.
- An allocator must keep bookkeeping data to track which parts of memory are available and which are in use. Traversing and updating these bookkeeping structures incurs overhead (especially if the allocator must be synchronized to be safely used across multiple threads).
- When data ends up stored in many scattered allocations, this lack of data locality often leads to more cache misses.
- Making many small allocations may fragment the available memory pool, which induces more bookkeeping overhead and reduces the size and number of large contiguous blocks that are available for future allocations.
- Like allocations, deallocations also incur bookkeeping costs and may trigger system calls.

To minimize allocation costs, the most effective strategy is to use "arena allocators", which effectively allow you to group your allocations. *(See this "Allocator overview" for the DOTS Collections package.)*

When using a garbage collector, there's also the major cost of the garbage collector periodically scanning memory to find and free the allocations which are no longer in use. The more allocations you make, the more frequently the garbage collector will have to run, so for your managed objects in C#, consider object pooling. *(See "Use object pooling to boost performance of C# scripts in Unity".)*

Are you using too many strings?

Excessive use of strings can drain performance for a few reasons:

- Strings come in all sizes, so they generally must be heap allocated.
- Comparing two strings may require traversing their full lengths.
- If strings in your language are immutable, then operations which slice and manipulate a string must allocate memory and copy data to make a new string rather than simply update the original string in-place.

To minimize string costs:

- Avoid creating many intermediate strings.
- For lookup keys and ids, prefer using integers instead of strings.
- When concatenating pieces together to make a string whose final size is known beforehand, preallocate the required memory for the full string up front. If the final size is not known beforehand, use a StringBuilder.

Are you taking full advantage of the CPU?

Can your code take better advantage of multi-core CPUs?

Are you using each CPU core efficiently?

Although single-threaded programming can be considered the natural 'default' way to program, it may leave a lot of potential performance on the table with today's multi-core CPUs. This is especially the case in games, where much of the heavy work lends itself to task concurrency or task parallelism.

Achieving the best performance, of course, also requires taking full advantage of each individual CPU core. For example, computation throughput can be greatly increased on an individual core by using SIMD instructions.

> **Note:** The DOTS job system makes multi-threading easy, and the Burst compiler can generate code with SIMD "vectorization" and other optimizations.

Performance checklist for DOTS

Performance considerations specific to DOTS.

Is your code Burst compiled?

In many cases, Burst-compiled code will be 2-5x faster than the same code compiled with IL2CPP, and it's common for Burst-compiled code to be 10-100x faster than the same code compiled with Mono. Using Burst is often the simplest, most effective optimization you will make, particularly for your computationally heavy code.

> **Note:** If you attempt to Burst-compile code that is not Burst compatible, the compiler will log an error in the Unity console. Unlike other compiler errors, these errors do not persist when you clear the console, so it's sometimes easy to not notice when Burst compilation fails. To check if a piece of code has actually been compiled with Burst, you can look in the Burst inspector window. Also, in the Unity profiler timeline view, code that is Burst-compiled is shown in green and has "Burst" in the label.

> **Note:** Burst does come with a notable downside: longer compilation times. Consequently, you shouldn't necessarily Burst-compile everything you possibly can whether it needs it or not. *(See this forum post.)*

Are your jobs scheduled efficiently?

Do your jobs have unnecessary dependencies?

The job system guarantees that a job will not begin execution until after all of its dependencies have themselves finished execution. While the job safety checks will catch most cases where you schedule a job without all of its required dependencies, the safety checks do *not* catch cases where you schedule a job with *unnecessary* dependencies. Therefore, you may end up

with jobs which wait unnecessarily for other jobs, and this could lead to cases where CPU cores needlessly sit idle.

In Entities, the way systems pass job handles may lead to unnecessary job dependencies:

1. A system receives job dependencies from all other systems that access any of the same component types. For example, a system that accesses component type `Foo` will receive job dependencies from all other systems that also access component type `Foo`.
2. However, the jobs scheduled in these systems do not necessarily themselves access `Foo`, in which case they shouldn't necessarily have to depend upon each other, and yet, all jobs of both systems will end up as each other's dependencies.

These unnecessary dependencies are not always a notable bottleneck, but when it is a problem, a workaround is to re-organize the code of the affected systems, splitting them into additional separate systems or consolidating them into fewer. Alternatively, you can pass the relevant job handles across the systems manually instead of through the standard `SystemState.Dependency` property.

Is the main thread wasting time waiting for jobs to complete? Could some jobs be scheduled earlier?

When the main thread calls `Complete()` on a job's handle, `Complete()` will not return until the job has finished execution. Ideally, the job has already finished execution by the time you make the call, so `Complete()` can then just return immediately.

To minimize main thread wait time, you should put off completing each job as long as possible. In cases where the `Complete()` call can't be moved back further, you can instead perhaps:

- schedule the job earlier, thus reducing or eliminating the `Complete()` wait time
- move up the scheduling of other jobs to before the `Complete()` call, thus giving the cores more work to do while the main thread waits

In fact, scheduling jobs as early as possible is a good strategy in general, as it helps minimize the time the cores sit waiting idle when they otherwise could be doing useful work.

Are data conflicts preventing jobs from executing concurrently?

If two jobs access any of the same data (meaning either they access the same collections or the same entity component types), then one of the two jobs must run and finish execution before the other. This can be guaranteed by having the first job either:

- be completed before scheduling the second job
- or be declared as a dependency when scheduling the second job

Because job completion may block the main thread, declaring a dependency is generally preferred. In either case, the jobs cannot run concurrently with each other.

In some scenarios, you can get better core utilization by splitting up jobs into smaller jobs which each use only a subset of the data. This potentially opens up opportunities for some of the smaller jobs to execute concurrently with each other. For example, if a job accesses two arrays, X and Y, then it cannot run concurrently with any other jobs that access either X or Y; if the work is split into one job that accesses only X and another job which accesses only Y, these smaller jobs potentially have fewer conflicts with other jobs and so may allow for more concurrent execution of the work.

Can some job data be declared read only?

If data accessed by two jobs is declared as 'read only' in both jobs, then the shared data does not represent a data conflict, and the job safety checks will not object if the two jobs are scheduled to run concurrently. Furthermore, randomly indexing an array or list in a parallel job is only allowed by the job safety checks if the array or list is declared read only in the job.

For both of the above reasons, whenever a collection or component type is only read in a job, you should declare it as read only. *(See the ReadOnly attribute.)*

For some jobs, it may pay off to defer the writes if doing so allows the data to be declared read only and thereby allows the job to run in parallel or concurrently with other jobs. For example, if a job writes to an array that is read by another job, then the two jobs cannot be scheduled concurrently. If, however, the writing job records changes for the array in a separate buffer instead of modifying the actual array directly, then the job can declare the array as read only, and then the two jobs *can* be scheduled concurrently. After both jobs finish, the changes recorded in the separate buffer can be copied into the original array. (Just understand that whether this trick actually improves performance depends upon a lot of factors: what proportion of the array gets modified, the relative running time of the two jobs, how busy the worker threads are with other jobs, *etc.* As always, be sure to profile to verify your optimizations.)

Can some single-threaded jobs be parallelized?

Are you parallelizing jobs that should be single-threaded?

If a job is scheduled as a parallel job, then its work is split into batches which are processed concurrently on separate threads. Some jobs, however, cannot be parallelized because splitting the work into concurrent batches would create internal race conditions. As just discussed in the prior section, you can sometimes rectify this by changing how you mutate the data. In other cases, you may be able to split up the job to isolate the un-parallelizable part in its own single-threaded job while the rest can be parallelized.

Just keep in mind that parallelization isn't always a performance win: due to memory contention, a parallel solution may end up using more CPU cores only to perform no better or even worse than a single-threaded solution. Even when this is not the case, you should consider whether a parallel job might be better off running single-threaded concurrently alongside other single-threaded jobs rather than taking up all of the CPU cores itself.

Are you paying for excessive structural changes of entities?

In Entities, a "structural change" is any operation that creates or destroys an entity or moves an existing entity in memory. Adding and removing components of an entity requires moving the entity, so they are structural changes operations.

Generally, dozens or even hundreds of structural changes in a single frame isn't a big deal, but the costs can add up to be significant, especially when you get into the thousands.

A common simple mistake is to unnecessarily add or remove components one-by-one:

- The methods for adding and removing components which take a `ComponentTypeSet` rather than just an individual component type will move the entity just once rather than one time for each component type.
- The methods for adding and removing components which take an EntityQuery will affect every entity matching the query instead of just a single entity. These operations copy the affected entities *en masse* rather than one-by-one.
- When creating many entities with the same set of component types, you should first create one entity with all of the desired components and then call `EntityManager.Instantiate()` to copy it. This is much cheaper than creating all of the entities first and then immediately adding or removing components from them individually.

For more discussion of how to minimize structural changes and the tradeoffs involved, see "StateChange" in the samples and this explainer video.

Are you using entities where you should instead be using arrays or other data structures?

As discussed in another document, entities are not necessarily the optimal data structure for everything.

Are you doing too many random lookups of entities by entity ID?

Thanks to the archetype- and chunk-based memory structure of Unity's ECS implementation, the components of entities matching a query can be traversed in a way that avoids incurring many expensive cache misses. However, accessing a specific entity by its entity ID is effectively a "random lookup" (a random jump to some place in memory) and therefore is very likely to trigger a cache miss. A single random lookup is no issue, and dozens or hundreds per frame is still probably fine, but thousands or tens of thousands may be a problem.

Of course, there are naturally many cases in typical game code where various entities have relationships with other arbitrary entities and so must reference each other by ID. For example, a Monster entity may have a designated Villager entity that it is currently chasing, so the Monster must store the Villager entity's ID. Naturally then, the Monster logic will, at certain points, need to perform a random lookup of each Monsters' referenced Villager. Perhaps, with some clever thinking, the required number of random lookups for this logic can be minimized, but it's unlikely in such cases that they can be avoided entirely.