Jinx Performance

Introduction

Because Jinx is targeted at real-time environments such as videogames, it's important for developers to have a realistic assessment of Jinx's overall performance characteristics. This paper presents the results of a performance test that exercises a wide range of features in various threaded environments, and on several different machine types.

Performance-Related Features

Jinx features a number of features specifically designed to help improve performance in real-time applications.

Thread-Safe Scripting

Jinx is designed to safely execute scripts in arbitrary threads. Because scripts naturally execute as co-routines, there is a minimal dependency on global resources, except when accessing library-wide functionality, such as getting or setting a property. As such, typical scripts generally do not suffer from much thread contention, and scale well on multiple cores.

This ensures that your own code can use Jinx scripts in a threaded environment without the use of performance-killing global locks.

Allocation Control

Jinx allows the user to supply a custom allocator, potentially enabling better performance than with the default system allocator, or simply enabling you to keep better track of your scripting allocations.

Jinx also utilizes internal per-object memory arenas to minimize the number of times the global allocator must be called, thereby improving performance.

Performance APIs

Jinx provides two API calls, `IRuntime::GetScriptPerformanceStats()` and `Jinx::GetMemoryStats()`, used for retrieving performance and memory stats respectively. This can help to provide runtime insights for both memory and CPU use to ensure Jinx stays within acceptable performance boundaries. You can see more details of these functions and the data they return in the online documentation.

Performance Tests

We conduct some synthetic benchmarks on three different machines, each running a different OS, in order to get a realistic idea of Jinx's performance characteristics. The performance test is included as part of the standard Jinx distribution, and is called *PerfTest*.

Machine One

- Type: 2009 Desktop PC
- CPU: 3.20GHz Intel Core i7 CPU 960 4 Cores, 8 HW Threads
- OS: Windows 10

```
--- Performance (1 thread) ---
Total run time: 1.826242 seconds
Total script execution time: 1.768427 seconds
Number of scripts executed: 340000 (186174 per second)
Number of scripts completed: 40000 (21902 per second)
Number of instructions executed: 23080000 (12.64M per second)

--- Performance (2 threads) ---
Total run time: 0.931908 seconds
Total script execution time: 1.796369 seconds
Number of scripts executed: 340000 (364842 per second)
Number of scripts completed: 40000 (42922 per second)
Number of instructions executed: 23080000 (24.77M per second)

--- Performance (3 threads) ---
Total run time: 0.645105 seconds
Total script execution time: 1.862327 seconds
Number of scripts executed: 340000 (527045 per second)
Number of scripts completed: 40000 (62005 per second)
Number of instructions executed: 23080000 (35.78M per second)

--- Performance (4 threads) ---
Total run time: 0.507980 seconds
Total script execution time: 1.915656 seconds
Number of scripts executed: 340000 (669317 per second)
Number of scripts completed: 40000 (78743 per second)
Number of instructions executed: 23080000 (45.43M per second)

--- Performance (5 threads) ---
Total run time: 0.538896 seconds
Total script execution time: 2.375794 seconds
Number of scripts executed: 340000 (630919 per second)
Number of scripts completed: 40000 (74225 per second)
Number of instructions executed: 23080000 (42.83M per second)

--- Performance (6 threads) ---
Total run time: 0.516185 seconds
Total script execution time: 2.689089 seconds
Number of scripts executed: 340000 (658678 per second)
Number of scripts completed: 40000 (77491 per second)
Number of instructions executed: 23080000 (44.71M per second)
```

```
--- Performance (7 threads) ---
Total run time: 0.466289 seconds
Total script execution time: 2.902782 seconds
Number of scripts executed: 340000 (729160 per second)
Number of scripts completed: 40000 (85783 per second)
Number of instructions executed: 23080000 (49.50M per second)

--- Performance (8 threads) ---
Total run time: 0.427553 seconds
Total script execution time: 3.232820 seconds
Number of scripts executed: 340000 (795223 per second)
Number of scripts completed: 40000 (93555 per second)
Number of instructions executed: 23080000 (53.98M per second)
```

Machine Two

- Type: 2012 Mac Mini
- CPU: 2.5GHz Intel Core i5 2 Cores, 4 HW Threads
- OS: macOS “Catalina”

```
--- Performance (1 thread) ---
Total run time: 2.218090 seconds
Total script execution time: 2.164140 seconds
Number of scripts executed: 340000 (153285 per second)
Number of scripts completed: 40000 (18033 per second)
Number of instructions executed: 23080000 (10.41M per second)

--- Performance (2 threads) ---
Total run time: 1.150211 seconds
Total script execution time: 2.229414 seconds
Number of scripts executed: 340000 (295598 per second)
Number of scripts completed: 40000 (34776 per second)
Number of instructions executed: 23080000 (20.07M per second)

--- Performance (3 threads) ---
Total run time: 1.055072 seconds
Total script execution time: 3.036598 seconds
Number of scripts executed: 340000 (322252 per second)
Number of scripts completed: 40000 (37912 per second)
Number of instructions executed: 23080000 (21.88M per second)

--- Performance (4 threads) ---
Total run time: 0.953081 seconds
Total script execution time: 3.690789 seconds
Number of scripts executed: 340000 (356737 per second)
Number of scripts completed: 40000 (41969 per second)
Number of instructions executed: 23080000 (24.22M per second)
```

Machine Three

- Type: 2016 Mini Desktop PC
- CPU: 3.2GHz Intel Core i5 6500 4 Cores, 4 HW Threads

- Ubuntu 20.04

```
--- Performance (1 thread) ---
Total run time: 1.034078 seconds
Total script execution time: 1.008879 seconds
Number of scripts executed: 340000 (328795 per second)
Number of scripts completed: 40000 (38681 per second)
Number of instructions executed: 23080000 (22.32M per second)

--- Performance (2 threads) ---
Total run time: 0.572235 seconds
Total script execution time: 1.092592 seconds
Number of scripts executed: 340000 (594162 per second)
Number of scripts completed: 40000 (69901 per second)
Number of instructions executed: 23080000 (40.33M per second)

--- Performance (3 threads) ---
Total run time: 0.417933 seconds
Total script execution time: 1.181758 seconds
Number of scripts executed: 340000 (813527 per second)
Number of scripts completed: 40000 (95709 per second)
Number of instructions executed: 23080000 (55.22M per second)

--- Performance (4 threads) ---
Total run time: 0.346926 seconds
Total script execution time: 1.297072 seconds
Number of scripts executed: 340000 (980036 per second)
Number of scripts completed: 40000 (115298 per second)
Number of instructions executed: 23080000 (66.53M per second)
```

Analysis

Jinx single-threaded performance ranges from 10.41 MIPS (Millions of Instructions Per Second) to 22.32 MIPS in this test on what would likely be considered low range PC gaming hardware in 2021. As such, this offers a reasonable worst-case performance benchmark for videogame projects targeting reasonably modern hardware.

Real World Performance Estimation

How does this translate to real world performance?

We can makes some very broad estimations based on these results. For our worst-case scenario estimates, let's assume our target machine can execute 10 MIPS per core, and that our scripting will all occur on the main thread.

In our test suite, the four test scripts compile to a total of 510 bytecode instructions from 99 lines of source code, resulting in an average of 5.1 instructions per line of source code.

Thus, 15 MIPS translates to roughly 4.41 million lines of scripting executed per second, or approximately 73,528 lines of scripting executed within 1/60th of a second. We now have a general baseline of what a single low-end CPU core can handle per frame in the context of a game running 60 FPS.

Obviously, we can't allow scripting to monopolize the CPU, so let's limit the scripting budget to 5% of a single core. This leaves us with an estimated budget of 3675 lines of scripting we can execute per frame while not significantly impacting the performance of a single core.

Jinx scripts are designed to run asynchronously, and as such, may employ scripting that waits for specific conditions to occur before continuing execution. In this scenario, a game could easily be running many dozens of scripts simultaneously, so long as a significant portion of them are in a waiting state at any given time. This is analogous to the efficiency of threads when they are waiting for a signal to resume execution.

In contrast, if a script is performing long, intense computations of any sort on each frame, a single script could easily exceed the allotted per-frame budget. Fortunately, Jinx can put a limit on a single script's maximum instruction count, effectively throttling it. This means the script will execute over a number of frames, and as such, should not negatively impact the CPU budget on any given frame in particular.

Threaded Performance

You can see that Jinx script execution scales in total MIPS fairly well with the number of cores it runs on, but performance benefits tend to drop off sharply when scaling up beyond the number of physical cores and into the range of hardware threads (in two of our test cases, 2 HW threads exist per core). In the case of macOS test and its Dual Core processor, per-thread performance drops significantly once the number of threads exceeds the number of physical cores.

Nonetheless, this demonstrates a practical solution to the potential issue of too many scripts saturating the game's primary thread, provided the native functions Jinx calls are also written in a thread-safe manner. By moving execution to another core, it becomes possible to execute significantly more scripts. How many more, of course, is simply a function of how much of a CPU budget is given to them.

Conclusion

We see in this paper how Jinx can easily execute many dozens concurrent scripts on modest hardware in real-time without overly taxing hardware, providing said scripts are designed to run in an asynchronous-friendly manner, using the wait instruction to amortize the CPU load over time.

Moreover, due to the thread-safe nature of Jinx scripts, it's practical to offload script execution to background threads or a thread pool, ensuring better scaling on modern multi-core processors.

Finally, Jinx provides a simple method of limiting the instruction count of any given script per execution call (typically once per frame), and so can ensure that neither heavy loads nor badly designed scripts can negatively impact the overall frame rate or cause hitching – an important consideration for real-time applications.

Generally speaking, Jinx is probably not a suitable candidate to write the bulk of your game's low-level logic in, as it imposes too much runtime overhead for that. Instead, it is best suited to providing asynchronous control over things such as in-game AI agents, one-off scripted in-game events, quest tracking, engine-specific tasks (audio, cinematics, world events), and other tasks that can benefit from a high-level in-game scripting system.