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.

Built-In Allocator

Jinx utilizes its own block allocator designed to prioritize efficiency for small, frequent allocations, as is typical of scripting requirements. Additionally, it makes use of thread-local storage pools to ensure minimal contention between scripts executing independently on different threads.

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: 2.600306 seconds
Total script execution time: 2.542499 seconds
Number of scripts executed: 340000 (130753 per second)
Number of scripts completed: 40000 (15382 per second)
Number of instructions executed: 22880000 (8.80M per second)

--- Performance (2 threads) ---
Total run time: 1.472654 seconds
Total script execution time: 2.864254 seconds
Number of scripts executed: 340000 (230875 per second)
Number of scripts completed: 40000 (27161 per second)
Number of instructions executed: 22880000 (15.54M per second)

--- Performance (3 threads) ---
Total run time: 1.045632 seconds
Total script execution time: 3.029988 seconds
Number of scripts executed: 340000 (325162 per second)
Number of scripts completed: 40000 (38254 per second)
Number of instructions executed: 22880000 (21.88M per second)

--- Performance (4 threads) ---
Total run time: 0.828761 seconds
Total script execution time: 3.170520 seconds
Number of scripts executed: 340000 (410251 per second)
Number of scripts completed: 40000 (48264 per second)
Number of instructions executed: 22880000 (27.61M per second)

--- Performance (5 threads) ---
Total run time: 0.813444 seconds
Total script execution time: 3.757712 seconds
Number of scripts executed: 340000 (417975 per second)
Number of scripts completed: 40000 (49173 per second)
Number of instructions executed: 22880000 (28.13M per second)

--- Performance (6 threads) ---
Total run time: 0.819450 seconds
Total script execution time: 4.307143 seconds
Number of scripts executed: 340000 (414912 per second)
Number of scripts completed: 40000 (48813 per second)
Number of instructions executed: 22880000 (27.92M per second)
```

```
--- Performance (7 threads) ---
Total run time: 0.744149 seconds
Total script execution time: 4.683302 seconds
Number of scripts executed: 340000 (456897 per second)
Number of scripts completed: 40000 (53752 per second)
Number of instructions executed: 22880000 (30.75M per second)

--- Performance (8 threads) ---
Total run time: 0.678510 seconds
Total script execution time: 5.223381 seconds
Number of scripts executed: 340000 (501097 per second)
Number of scripts completed: 40000 (58952 per second)
Number of instructions executed: 22880000 (33.72M per second)
```

Machine Two

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

```
--- Performance (1 thread) ---
Total run time: 2.787829 seconds
Total script execution time: 2.723717 seconds
Number of scripts executed: 340000 (121958 per second)
Number of scripts completed: 40000 (14348 per second)
Number of instructions executed: 22880000 (8.21M per second)

--- Performance (2 threads) ---
Total run time: 1.459643 seconds
Total script execution time: 2.825651 seconds
Number of scripts executed: 340000 (232933 per second)
Number of scripts completed: 40000 (27403 per second)
Number of instructions executed: 22880000 (15.68M per second)

--- Performance (3 threads) ---
Total run time: 1.506838 seconds
Total script execution time: 4.295106 seconds
Number of scripts executed: 340000 (225638 per second)
Number of scripts completed: 40000 (26545 per second)
Number of instructions executed: 22880000 (15.18M per second)

--- Performance (4 threads) ---
Total run time: 1.867430 seconds
Total script execution time: 7.092506 seconds
Number of scripts executed: 340000 (182068 per second)
Number of scripts completed: 40000 (21419 per second)
Number of instructions executed: 22880000 (12.25M per second)
```

Machine Three

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

- Ubuntu 16

```
--- Performance (1 thread) ---
Total run time: 1.511879 seconds
Total script execution time: 1.473931 seconds
Number of scripts executed: 340000 (224885 per second)
Number of scripts completed: 40000 (26457 per second)
Number of instructions executed: 22880000 (15.13M per second)

--- Performance (2 threads) ---
Total run time: 0.865259 seconds
Total script execution time: 1.647602 seconds
Number of scripts executed: 340000 (392946 per second)
Number of scripts completed: 40000 (46228 per second)
Number of instructions executed: 22880000 (26.44M per second)

--- Performance (3 threads) ---
Total run time: 0.667076 seconds
Total script execution time: 1.887071 seconds
Number of scripts executed: 340000 (509687 per second)
Number of scripts completed: 40000 (59963 per second)
Number of instructions executed: 22880000 (34.30M per second)

--- Performance (4 threads) ---
Total run time: 0.612747 seconds
Total script execution time: 2.249341 seconds
Number of scripts executed: 340000 (554878 per second)
Number of scripts completed: 40000 (65279 per second)
Number of instructions executed: 22880000 (37.34M per second)
```

Analysis

Jinx single-threaded performance ranges from 8.21 MIPS (Millions of Instructions Per Second) to 15.13 MIPS in this test on what would likely be considered low to mid-range PC gaming hardware in 2017. 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 8 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 340 bytecode instructions from 99 lines of source code, resulting in an average of 3.4 instructions per line of source code.

Thus, 8 MIPS translates to roughly 2.35 million lines of scripting executed per second, or approximately 39,215 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 3% of a single core. This leaves us with an estimated budget of 1176 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 actually drops once threads exceed 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.