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: 3.314056 seconds
Total script execution time: 3.244217 seconds
Number of scripts executed: 340000 (102593 per second)
Number of scripts completed: 40000 (12069 per second)
Number of instructions executed: 22880000 (6.90M per second)

--- Performance (2 threads) ---
Total run time: 1.752988 seconds
Total script execution time: 3.415886 seconds
Number of scripts executed: 340000 (193954 per second)
Number of scripts completed: 40000 (22818 per second)
Number of instructions executed: 22880000 (13.05M per second)

--- Performance (3 threads) ---
Total run time: 1.237908 seconds
Total script execution time: 3.583193 seconds
Number of scripts executed: 340000 (274656 per second)
Number of scripts completed: 40000 (32312 per second)
Number of instructions executed: 22880000 (18.48M per second)

--- Performance (4 threads) ---
Total run time: 1.020461 seconds
Total script execution time: 3.874900 seconds
Number of scripts executed: 340000 (333182 per second)
Number of scripts completed: 40000 (39197 per second)
Number of instructions executed: 22880000 (22.42M per second)

--- Performance (5 threads) ---
Total run time: 1.010381 seconds
Total script execution time: 4.508929 seconds
Number of scripts executed: 340000 (336506 per second)
Number of scripts completed: 40000 (39589 per second)
Number of instructions executed: 22880000 (22.64M per second)

--- Performance (6 threads) ---
Total run time: 0.945903 seconds
Total script execution time: 5.203672 seconds
Number of scripts executed: 340000 (359444 per second)
Number of scripts completed: 40000 (42287 per second)
Number of instructions executed: 22880000 (24.19M per second)
```

```
--- Performance (7 threads) ---
Total run time: 0.846344 seconds
Total script execution time: 5.597741 seconds
Number of scripts executed: 340000 (401728 per second)
Number of scripts completed: 40000 (47262 per second)
Number of instructions executed: 22880000 (27.03M per second)

--- Performance (8 threads) ---
Total run time: 0.807679 seconds
Total script execution time: 6.141680 seconds
Number of scripts executed: 340000 (420959 per second)
Number of scripts completed: 40000 (49524 per second)
Number of instructions executed: 22880000 (28.33M 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: 3.370361 seconds
Total script execution time: 3.300054 seconds
Number of scripts executed: 340000 (100879 per second)
Number of scripts completed: 40000 (11868 per second)
Number of instructions executed: 22880000 (6.79M per second)

--- Performance (2 threads) ---
Total run time: 1.724174 seconds
Total script execution time: 3.348469 seconds
Number of scripts executed: 340000 (197195 per second)
Number of scripts completed: 40000 (23199 per second)
Number of instructions executed: 22880000 (13.27M per second)

--- Performance (3 threads) ---
Total run time: 1.902072 seconds
Total script execution time: 5.399718 seconds
Number of scripts executed: 340000 (178752 per second)
Number of scripts completed: 40000 (21029 per second)
Number of instructions executed: 22880000 (12.03M per second)

--- Performance (4 threads) ---
Total run time: 1.875996 seconds
Total script execution time: 7.228067 seconds
Number of scripts executed: 340000 (181237 per second)
Number of scripts completed: 40000 (21322 per second)
Number of instructions executed: 22880000 (12.20M 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: 1.758575 seconds
Total script execution time: 1.718695 seconds
Number of scripts executed: 340000 (193338 per second)
Number of scripts completed: 40000 (22745 per second)
Number of instructions executed: 22880000 (13.01M per second)

--- Performance (2 threads) ---
Total run time: 0.955246 seconds
Total script execution time: 1.838553 seconds
Number of scripts executed: 340000 (355929 per second)
Number of scripts completed: 40000 (41874 per second)
Number of instructions executed: 22880000 (23.95M per second)

--- Performance (3 threads) ---
Total run time: 0.753214 seconds
Total script execution time: 2.112014 seconds
Number of scripts executed: 340000 (451399 per second)
Number of scripts completed: 40000 (53105 per second)
Number of instructions executed: 22880000 (30.38M per second)

--- Performance (4 threads) ---
Total run time: 0.649953 seconds
Total script execution time: 2.431625 seconds
Number of scripts executed: 340000 (523114 per second)
Number of scripts completed: 40000 (61542 per second)
Number of instructions executed: 22880000 (35.20M per second)
```

Analysis

Jinx single-threaded performance ranges from 6.8 MIPS (Millions of Instructions Per Second) to 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 6 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 code.

Thus, 6 MIPS translates to roughly 1.75 million lines of scripting executed per second, or approximately 30,000 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 it to a total of 3% of a single core. This leaves us with an approximate budget of 875 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 run hundreds 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, a single script could easily exceed the allotted 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 thousands of lightweight scripts simultaneously, or hundreds of more complex ones.

Conclusion

We see in this paper how Jinx can easily execute several hundred 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.