Benchmarking the Nintendo Switch

Edward Chen, Evan Laufer, Miranda Go

1 Introduction

An operating system plays a vital role in resource management, time-sharing, offering protection, and much more. The operating system manages all of the software and hardware running on the system, and it is a crucial component to the performance of the system.

In this paper, we explore different benchmarking methods and results for Atmosphere, a modified version of Horizon OS, the operating system that runs on the Nintendo Switch. All benchmarking code was written in C++ and compiled using gcc 9.2.0 without optimizations. In order to access the Atmosphere system API, we used the libnx library with devkitpro v3.0.3. In total, we estimate the time spent on set-up, development, and running our benchmarking tests took approximately 90 hours.

There were several challenges posed when trying to benchmark the Switch. First, the default operating system, Horizon OS, installed on the Switch does not allow arbitrary code execution, necessitating the use of Atmosphere to let us run code. Second, the system API is not very well documented, making writing and debugging certain programs difficult. Third, the operating system has some key differences to the monolithic kernels we are familiar with, making certain benchmarks difficult or impossible. The structure of Horizon and Atmosphere, as well as the challenges we faced, will be discussed in further depth in the next section. [2]

2 Background

2.1 Setup

In order to benchmark the Switch, our first task was figuring out how to run arbitrary code on the Switch so we could run benchmarking tools. The default operating system on the switch, Horizon OS, has no shell and can only run applications on Nintendo game cartridges or from the Nintendo E-Shop. Therefore, the only way to achieve arbitrary code execution is to somehow exploit the system, and install a custom operating system (or firmware). Lucky for us, tools have been previously written to run custom firmware and programs (called *Homebrew* within the online community). The steps for installing and running Homebrew are: 1. Finding a vulnerable Switch [1], 2. Pushing a custom recovery mode, 3. Booting into Atmosphere, the custom firmware, and 4. Using Atmosphere to load and run our code. [4] [3]

The current system used by others to install custom firmware is called CVE-2018-6242 ("Fusée Gelée") [7], and is a hardware flaw in the original Tegra processor. In short, the processor supports a recovery mode (called RCM) in the bootloader which is used for recovery and system maintenance by Nintendo, and provides a USB interface to interact with. The RCM contains a buffer overflow vulnerability that allows an attacker to copy arbitrary data to the stack, and gain control of the system. This exploit cannot be patched by software, and thus certain Switch systems that were manufactured before a certain date are still vulnerable. There is a simple website that says if your system is vulnerable with its serial number. Of the three Switches we have, one is vulnerable.

Once we found a vulnerable Switch, we performed the exploit and pushed a custom bootloader, which allowed us to perform a system backup and launch Atmosphere. The first step in this process was to enter RCM, which could only be accessed by pressing the the home button, the volume up button, and the power button at the same time while booting. The only catch is the Switch itself has no physical home button, and instead emulates one on the joy-con. The way to "press" the hardware home button is to ground pin 10 on the right joycon connector, which we did by bridging it to pin 0 using a wire jig (Figure 1). Once we entered RCM mode, the rest was fairly simple, using the "Fusee Gelee" python script to push the Hekate custom bootloader over USB.

After launching the custom bootloader, launching Atmosphere is trivial and only requires copying some files to an SD card. Once in Atmosphere, custom applications can be run through the "Homebrew Menu". It is important to note that these applications are not the same as official Nintendo applications, in that Atmosphere contains a custom loader to run them. Certain system calls on the Switch reference metadata that only official Nintendo applications have, such as the "title-id" field. This mostly only affected our ability to perform benchmarks that required process creation or process context-switching.

One caveat to this is we installed Atmosphere on the SD card, as opposed to overwriting the internal system disk. This should have a minimal effect on most metrics, however it may have some unknown impact, as we could not compare the benchmarks to a Switch running off the internal storage.

2.2 Horizon and Atmosphere Architecture

Atmosphere does not modify most of the Horizon system, and as such the system architecture between both is nearly the same. As such, the following information applies to both systems. In general, Horizon follows a micro-kernel architecture. The kernel itself has a fairly small amount of functionality, with most operating system tasks being handled in external processes. These processes may be accessed through *services* that they can export. A few notable services are the file system service, process manager service,

(a) Nintendo Switch and Wire Jig

(b) Wire jig placed inside right controller slot (aerial view)

(c) Wire jig

(d) Wire jig placed inside right controller slot (side view)

(e) Sending the payload to run Hekate, a custom Nintendo Switch bootloader

Figure 1: Setup Pictures

and sockets service. [2]

In order to use a service, a process first requests the service's handle from the kernel. With the handle, the process may use inter-process communication mechanisms exposed by the kernel to make requests to the service. The most important parameters that the IPC mechanism uses are message type and data buffers. The service may respond to the request with data, or perform an operation.

2.3 System Ticks

The ARM Cortex A57 CPUs that the Nintendo Switch uses have a control register that measures cycle counts. Unfortunately, access to that register is protected. However, the CPU also has a register that tracks a *system tick*. *System ticks* are an internal measure of time used by delays and timers, and on the Switch its frequency is set to 19.2 MHz. Because we found the the tick frequency to be very consistent and the cost to read the tick very low (simply reading a register), we used this to measure time within our benchmarks. The conversion between system ticks and time is simply $t = n/(19.2 * 1000000)$, where t is time in seconds and n is the number of system ticks. A downside of using system ticks is they are more coarse grained than cycles, so many trials of any very short operations must be run to see a noticeable difference in the number of ticks. [2]

3 Machine Description

Please reference Table 1

4 CPU Operations

4.1 Measurement Overhead

4.1.1 Overhead of Reading Time

Prediction: In this section, we are measuring the overhead of reading a system tick. We predict that the overhead of reading time for reading the system tick register is around 10 cycles. This is because we predict that it will cost 3 cycles to read the tick register, 4 cycles to store the value of the tick register onto the stack, and finally 3 additional cycles to re-read the tick register to get the value of the tick register. Since we do not know the relation of the cycles to ticks, we arbitrarily predict that it will take 1 system tick per read. Therefore we estimate that it will be 10 cycles per system tick.

Methodology: In order to measure the overhead, we performed two experiments. The first experiment was measuring the difference between read system tick calls after sleeping for 10 seconds. We ran 15 trials in total and averaged the results of the last 10 trials to remove warm-up costs. The second experiment was measuring consecutive calls with no other instructions between each read system

Model	NVIDIA Custom Tegra X1 (model T210)
Cores	4 ARM Cortex A57 Cores
Clock Rate	1.020 GHz
L1 Instruction Cache Size	48 KiB
L1 Data Cache	32 KiB
L2 Cache	2 MiB

(a) Processor Specification

Model	Samsung K4F6E304HB-MGCH
Size	4 GB (2 + 2 GB modules)
Bus	1600 MHz LPDDR4 (25.5 GB/s bandwidth)
Width	64 bit

(b) Memory Specification

Model	Samsung KLMBG2JENB−B041 or Toshiba THGBMHG8C2LBAIL
Type	eMMC 5.1
Size	32 GB
Cache Size	65536 bytes (Samsung) or 4096 (Toshiba)
Bus	PCIe

(c) Internal Disk Specification

Model	Broadcom BCM4356XKUBG
Protocol Specification	802.11ac
Speed	433 Mb/s one-stream 867 Mb/s two-stream

(d) Network Specification

Model	SanDisk Ultra Plus Micro SD Card
Type	Micro SD Card
Size	128 GB
Speed	10 MB/s

(e) External Disk Specification

Table 1: Nintendo Switch Specifications

tick call instruction. We averaged 20 consecutive reads over 1000 trials to see the average increase in system ticks over per 20 read system tick call instructions.

Results:

Trial #	# of Ticks
1	192000099
2	192000096
3	192000094
4	192000091
5	192000093
6	192000090
7	192000091
8	192000088
9	192000091
10	192000091

Table 2: Empty Loop Time

Mean: 19200092.4 Ticks
Standard Deviation: 3.040 Ticks

Analysis: From the results of the first experiment, we concluded that it was around 19200000 system ticks per second. This is consistent with the reference we found which measured that Nintendo Switch has a frequency of about 19200092 system ticks per second. Given that our processor runs at maximum 1 GHz, we assume that each system tick is roughly 52 cycles. From our second experiment, we found that the average system ticks read instruction per 20 instruction calls was about 3 system ticks. We additionally found that the the system tick was updated per 4 read system tick call instructions. Based on our prediction, we estimate each system tick call instruction around 10 cycles, giving us a lower bound prediction that each system tick is roughly 40 cycles. Since we are benchmarking the Nintendo Switch in terms of system ticks and the measurements are between 40 cycles to 52 cycles, the overhead per read is about 0.25 system ticks to 0.325 system ticks per cycle. The estimate of one cycle is about 0.0192 (1/52) to 0.025 (1/40) system ticks.

4.1.2 Overhead of Empty Loop

Prediction: In this section, we are measuring the overhead of an empty loop. We predict that the overhead of an empty loop is around 9 cycles. This is because we predict that it will cost 3 cycles to compare the loop condition, 3 cycles to branch to the top of the loop, and finally 3 additional cycles to increment the loop counter. From our overhead of reading time test, we can estimate that the loop overhead would be around 0.173 to 0.225 system ticks.

Methodology: In order to measure the overhead, we performed 110 trials and in each trial we ran an empty loop 1000 times. Finally, we discard the first 10 measurements to account for CPU warm-up and took the median time of the remaining 100 times of running the loop.

Results: Mean of Medians: 0.1519 Ticks (7.911 ns)
Standard Deviation of Median: 0.0003 Ticks (0.015 ns)

Analysis: From our measurements, we conclude that the overhead of running an empty loop is around 0.152 ticks. This is fairly close to our estimated prediction of the overhead of an empty loop. We attribute the effects of pipelining to the faster measurements compared to our original predictions.

Trial #	Median (*tick*)	Median (*ns*)
1	0.152	7.917
2	0.152	7.917
3	0.152	7.917
4	0.152	7.917
5	0.152	7.917
6	0.152	7.917
7	0.152	7.917
8	0.152	7.917
9	0.151	7.865
10	0.152	7.917

Table 3: Empty Loop Time

4.2 Procedure Call Overhead

4.2.1 Procedure Times per Iteration

Prediction: In this section, we are measuring the overhead of each procedure call given differing number of input parameters. We predict that the base procedure call overhead with zero parameters will be 11 cycles. This is because we estimate for a procedure call, this will induce 3 cycles for a branch instruction, 1 cycle for a nop instruction, and 7 cycles for a return instruction. For each additional parameter, we predict it will add an additional 3 cycles per parameter to account for the extra mov instructions of adding the parameter on to the stack. Therefore, in system ticks, we predict that the base procedure call overhead would be between 0.212 ticks and 0.275 ticks. Each additional parameter will add an additional 0.057 ticks.

Methodology: We ran 1010 trials in total. For each trial, we ran a loop 1000 times, calling the procedure call 10 times. To account for CPU warm up, we disposed the first 10 trials and took the median and mean of the remaining 1000 trials. The measurements show the number of ticks per procedure call.

Unfortunately for the procedure call with no arguments, we tried account for CPU warm up of this procedure call by recording the average times of the last 1000 trials. We also tried moving the procedure call to the end of the testing file after all of the other procedure calls with differing arguments. Finally, we even examined the assembly code of the procedure call to see if extra instructions were being call compared to the procedure calls containing input arguments. Out of all of these minor modifications, we were still unsuccessful in lowering the average time to call a procedure call with 0 arguments compared to procedure calls with arguments. We attribute this overhead with the underlying operating system and do not have a proper explanation for this phenomenon.

Results: Trial Measurements are in *ticks*

# of Args	Trial 1	Trial 2	Trial 3
0	0.118632	0.118634	0.118634
1	0.079111	0.079110	0.079129
2	0.079109	0.079115	0.079109
3	0.097929	0.097928	0.097928
4	0.105471	0.105469	0.105474
5	0.122414	0.122412	0.122410
6	0.139360	0.139355	0.139354
7	0.160060	0.160068	0.160051

# of Args	Trial 4	Trial 5	Trial 6
0	0.118630	0.118636	0.118633
1	0.079102	0.079099	0.079100
2	0.079115	0.079107	0.079112
3	0.097930	0.097932	0.097929
4	0.105469	0.105470	0.105466
5	0.122409	0.122410	0.122414
6	0.139358	0.139350	0.139348
7	0.160057	0.160054	0.160060

# of Args	Trial 7	Trial 8	Trial 9	Trial 10
0	0.118628	0.118647	0.118629	0.118632
1	0.079107	0.079106	0.079108	0.079108
2	0.079113	0.079102	0.079102	0.079106
3	0.097933	0.097931	0.097932	0.097930
4	0.105470	0.105467	0.105473	0.105468
5	0.122413	0.122414	0.122412	0.122416
6	0.139357	0.139354	0.139353	0.139353
7	0.160063	0.160061	0.160058	0.160061

# of Args	Avg (*ticks*)	Avg (*ns*)	Std Dev (*ticks*)	Std Dev (*ns*)
0	0.118633	6.179	0.0000053	0.000276
1	0.079108	4.120	0.0000080	0.000417
2	0.079109	4.120	0.0000045	0.000234
3	0.097930	5.101	0.0000017	0.000089
4	0.105469	5.493	0.0000024	0.000125
5	0.122412	6.376	0.0000021	0.000109
6	0.139354	7.258	0.0000034	0.000177
7	0.160059	8.336	0.0000045	0.000234

A graph of the results are shown in Figure 2.

Analysis: Increment Overhead of an Argument: 0.005918 Ticks (0.308 *ns*) Increase per additional argument in the procedure calls.

The trend of procedure call measurements was not what we expected. Specifically, the procedure call with no parameters should have taken the least amount of time. Without any optimizations, we expected that an increase in parameters would indicate a higher tick count since there would be an increase in total assembly instructions to put each parameter on the stack before calling the function. Since the procedure calls with no parameters is an outlier, we also calculated the tick increase per additional argument without the 0 argument procedure call, which is a 0.013492 tick (0.703 *ns*) increase per additional argument.

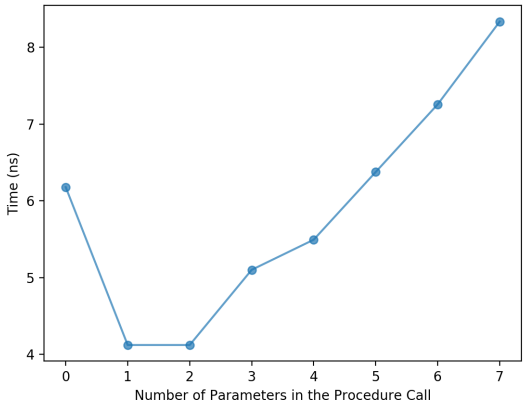

Figure 2: Average Time Per Procedure Call

4.3 System Call Overhead

4.3.1 System Call: GetSystemTick()

Prediction: In this section, we are trying to measure the time it will take to complete a system call. We predict that each system call will take around 25 cycles: 3 Cycles to move instruction into register, 11 Cycles for the function header, 8 Cycles to figure out the System Tick and load it, 3 Cycles to read the register at the end of the call. We also know that the system call would be crossing protection domains, which we estimate to take about 200-300 ns, which is about 5-6 Ticks. Adding that into our calculations, a system call would take roughly 5.48-6.48 ticks.

Methodology: To do so, we measured 10 trials of calling the system call GetSystemTick() 10000 times. For each trial, we divided results by 10000 to get the time for one system call.

Trial #	Time Per Loop Iter (*ticks*)	*ns*
1	6.5417	340.7
2	6.5387	340.6
3	6.5417	340.7
4	6.6321	345.4
5	6.5406	340.7
6	6.5426	340.8
7	6.5546	341.4
8	6.5414	340.7
9	6.5543	341.4
10	6.541	340.7

Results: Mean of Ticks Per Loop Iteration: 6.542 Ticks (340.7 ns)
Standard Deviation: 0.007 Ticks (0.365 ns)
System Call Without Loop Overhead: 0.6390 Ticks (33.3 ns)
A System call takes 55x longer than a procedure call with no parameters.

Analysis: From our results, it took on average 6.542 system ticks per system call. Taking into account the empty loop overhead of 0.152 ticks, this system call would be 6.390 ticks. Compared to our prediction of 5.48-6.48 system ticks, our results were within that range. In our prediction, we were not able to factor in the actual time for the assembly instructions for that specific system call (GetSystemtick()), which explains the slight variance in the results.

4.4 Task Creation Time

4.4.1 Creation of a Kernel Thread

Prediction: In this section, we are measuring the time it would take to create a kernel thread. We predict that the cost of creating a kernel thread would be greater than the cost it would take to make a system call because we would need to allocate the necessary data structures for this new thread.

Methodology: In order to measure the time it takes to create a kernel thread, we first created an empty thread function that just returned. For each trial, we then measured how long it took to create and run our empty thread 1010 times. The first 10 times were discard to account for CPU warm-up. The times were taken before the thread was created and after the thread has completely finished running. 10 trials were taken and averaged.

Results: Mean of Means: 558.78 Ticks (29.10 μs)
Standard Deviation of Mean: 1.685 Ticks (0.088 μs)
Mean of Medians: 545.33 Ticks (28.40 μs)
Standard Deviation of Median: 1.333 Ticks (0.069 μs)

Analysis: From our results, the average number of ticks to create a thread is about 559 ticks. Compared to our

prediction, the time to create a thread is significantly higher than making a system call by 875x.

Trial #	Mean (*ticks*)	μs	Median (*ticks*)	μs
1	562	29.27	549	28.59
2	561	29.22	548	28.54
3	558	29.06	544	28.33
4	557	29.01	544	28.33
5	558	29.06	546	28.44
6	558	29.06	547	28.49
7	560	29.17	545	28.39
8	558	29.06	545	28.39
9	558	29.06	545	28.39
10	557	29.01	544	28.33

4.4.2 Creation of a Process (Launching Photoviewer)

Prediction: In this section, we are trying to measure the overhead of creating a process. Unfortunately, we are unable to create our own processes for the Nintendo Switch, because we were unable to gain access to privileged system calls. We were able to launch existing processes through a defined service layer. Thus we predict that this cost of launching a process would induce extremely high overhead costs, on the factor of 10s to 100s of ticks.

Methodology: The process we decided to launch was to open the Photoviewer application. In order to measure the overhead cost of launching this process, we measured the time it took to launch then immediately kill a process. We ran 1010 trials, each trial containing a loop of 1000 iterations of creating, launching, and killing a process. We discarded the first 10 trials to account for CPU warm up.

Trial #	Mean (*ticks*)	ms	Median (*ticks*)	ms
1	486910	25.36	487270	25.38
2	484537	25.24	484804	25.25
3	484693	25.24	485282	25.28
4	484115	25.21	484415	25.23
5	486070	25.32	486703	25.35
6	484857	25.25	485475	25.29
7	484279	25.22	484976	25.26
8	486548	25.34	487040	25.36
9	486265	25.32	486846	25.36
10	486414	25.33	486875	25.36

Results: Mean of Means: 485468.8 Ticks (25.28 ms)
Mean of Medians: 485968.6 Ticks (25.31 ms) Standard Deviation of Mean: 1011.105 Ticks (0.053 ms)
Standard Deviation of Median: 1021.888 Ticks (0.053 ms)
Creating a process takes 867x longer than creating a kernel thread

Analysis: From our results, the average time to create a process was around 485968.6 ticks. We attribute this large overhead compared to thread creation to the inability to directly invoke the system call to create a process. Instead, the steps needed to create a process required communicating through the process manager service layer. This result is consistent with our prediction in that we would see a large overhead cost to create a process from these service calls.

4.5 Context Switch Time

4.5.1 Context Switching Between Producer and Consumer Kernel Threads

Prediction: Based on our references, a context switch should take about 5 to 7 microseconds. Converting that into system ticks, we predict the overhead of context switching should be about 96 to 134.4 ticks.

Methodology: In order to measure the overhead time of context switching, we created a producer thread that increments to 1010 and a consumer thread that also increments to 1010. The producer and consumer threads are bounded by a buffer of size 1 such that the two threads will ping pong. We measured the time between the producer unlock call to the consumer lock call. For each trial, we discarded the first 10 context switches to account for CPU warm up. 10 trials were taken.

Trial #	Mean (*ticks*)	μs	Median (*ticks*)	μs
1	57	2.968	56	2.917
2	57	2.968	56	2.917
3	57	2.968	55	2.865
4	56	2.917	56	2.917
5	57	2.968	56	2.917
6	56	2.917	55	2.865
7	56	2.917	55	2.865
8	57	2.968	56	2.917
9	57	2.968	56	2.917
10	57	2.968	56	2.917

Results: Mean of Means: 56.7 Ticks (2.953 μs)
Mean of Medians: 55.7 Ticks (2.901 μs)
Standard Deviation of Mean: 0.458 Ticks (0.024 μs)
Standard Deviation of Median: 0.458 Ticks (0.024 μs)

Analysis: From our results, the average ticks it takes to perform a context switch is about 55.7 to 56.7 ticks. This result is smaller than our predicted estimate. We believe this speed up is due to smaller data structures in the Nintendo Switch in each thread compared to Linux threads from our reference.

4.5.2 Process Context Switching

Unfortunately, we are unable to run our own code as a new process for the Nintendo Switch, because we were unable to gain access to privileged system calls. Nintendo protects process creation through a Process Manager service, and only applications that are packed by Nintendo and have a valid title ID may be launched. Although we were able to launch existing processes through a defined service layer,

we could not force process context switches between two processes nor write our scripts to measure this process context switch overhead cost. Thus we are unable to measure the overhead costs of process context switching.

5 Memory Operations

5.1 RAM Access Time

Latency of Accessing Main Memory, L1 Cache, and L2 Cache

Prediction: Assuming an L1 Cache access is around 3 clock cycles, we predict the latency for an individual L1 cache hit will be around 3 ns. Similarly, assuming an L2 cache access is around 10 clock cycles, we assume an L2 cache hit will be around 10 ns. Lastly, we assume a memory access will take around 100s of clock cycles, which would be around the scale of 100 ns.

For each memory read, we also update a pointer on the stack to track where in the array we are, which will take around 3 extra cycles. Therefore, we assume each L1 cache access will take from $0.115ns$ to $0.15ns$, each L2 cache access will take $0.25ns$ to $0.325ns$, and each RAM access will take $1.98ns$ to $2.575ns$.

Methodology: We measured the time it took to iterate and access an array 1000000 with various sizes and strides. The sizes went from 2^7 to 2^{22} and the strides went from 2^6 to 2^{10}. With each measurement, we first populated the array with void pointers where each pointer pointed to the address of the next array address to access (based on the stride). Then, we measured the time it took to iterate through 1000000 accesses through the populated array.

Results: A graph of the results are shown in Figure 3.

Analysis: First, the graph shows the latency jump at 32 Kib and 2 Mib, exactly as we expect from the known L1 Data Cache and L2 Cache sizes in the hardware specifications. We should also expect the latency amongst the strides to increase with stride size until we hit the cache-line size. The graph shows this stride size is 1024 bytes. From the technical documentation, the ARM Cortex A57 has 64 byte cache-lines. The discrepancy could be due to advanced cache-line prefetching behaviors.

In terms of latency time, we found the L1 cache took on average $8.13ns$, L2 cache took on average $30.45ns$, and the RAM access took on average $161.69ns$. The estimates are fairly close to our predictions; each of the predictions were on the estimated order of magnitude. We can account for variance in ram access time due to software overhead.

5.2 RAM Bandwidth

5.2.1 Reading from Memory

Prediction: In this section we measure the bandwidth of reading from memory. According to the hardware specifications for the RAM, we calculate the maximum bandwidth as $2 \text{ lines} * 8 \text{ bytes} * 1600 \text{ MHz} = 25.6\text{GB/s}$. We predict we will reach around that bandwidth.

Methodology: For each trial, we iterated and accessed a char array of size 2^{20} once with a stride of 2^8 and took the total time, which means there were 2^{12} access. So, the measurements shown is the total time for all the access divided by 2^{12} to calculate the time for one read.

5.2.2 Writing to Memory

Prediction: In this section we measure the bandwidth of reading from memory. We predict that the bandwidth will be about the same as reading memory, 25.5GB/s.

Methodology: For each trial, we iterated through a char array of size 2^{20} once with a stride of 2^8 and took the total time of writing one character into each spot, which means there were 2^{12} access. So, the measurements shown is the total time for all the writes divided by 2^{12} to calculate the time for one write.

Trial #	Read (GB/s)	Write (GB/s)
1	9.98	9.09
2	11.04	9.16
3	10.73	9.36
4	10.69	9.13
5	10.61	9.13
6	10.74	9.05
7	10.74	9.14
8	10.73	8.95
9	10.63	9.13
10	10.74	9.28

Results: Average Ticks per Read: $10.663GB/s$
Standard Deviation for Read: $0.253GB/s$
Average Ticks per Write: $9.142GB/s$
Standard Deviation for Write: $0.107GB/s$

Analysis: On average, our read bandwidth was $10.6GB/s$, which is a little less than half of our prediction. On average, our write bandwidth was $9.05GB/s$, which is about a third of our predicted bandwidth. This didn't meet our original prediction, but we assume one thread cannot saturate the entire bandwidth of the RAM bus, therefore the result is reasonable.

5.3 Page Fault Time

As of this time, we are unable to measure the time it takes for a page to be fetched or written to disk. We believe that when the application is loaded, all of its required pages and data information is loaded into memory. Currently each of our testing scripts is treated as a stand-alone applet or application and thus all of its required pages is stored in memory on load.

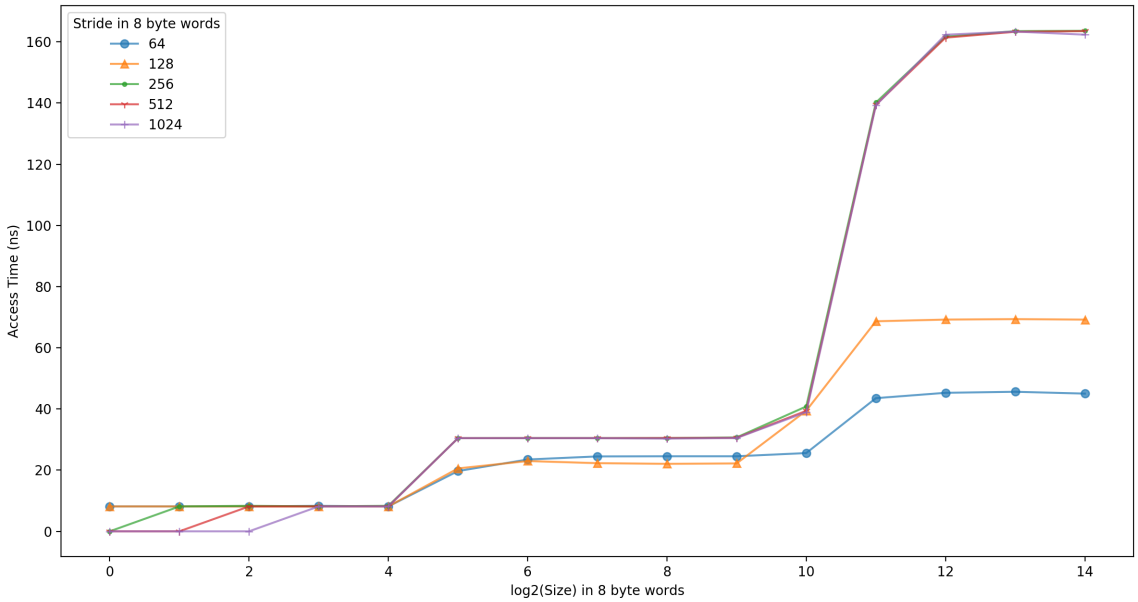

Figure 3: Memory Access Results

From our hardware specs, we are given 4 GB of main memory. We have tested the maximum amount of memory we can allocate for an application's heap and this is bounded at a little under 4 GB. Any attempts to allocate more memory results in a system fault and the Nintendo Switch crashing (did you know a Nintendo Switch could blue screen? Figure 4)

Additionally, we have tried measuring a fetching a page from the data section of the application by creating a large static global array and reading from this array. The results for each read is within 0 to 1 ticks, which symbolize that the data is already stored in memory and not in disk. This is again attributed to the operating system preloading the entire application into memory for running.

6 Machine Description - Dell XPS

The Dell operating system is Windows 10 Home. Please reference the remaining specs in Table 4

Trial #	Time (ms)
1	17
2	3
3	4
4	7
5	4
6	113
7	7
8	6
9	5
10	14
11	5
12	4
13	12
14	10
15	8
16	3
17	3
18	15
19	6
20	9

Average Time for Ping: 12.8 ms
Standard Deviation for Ping: 23.4 ms

7 Network: TCP

7.1 Ping Time

Methodology: In order to get a good estimate of our network speed, we initially pinged the Switch 20 times from the Dell XPS. We then averaged the results to find the average ping time.

Analysis: We believe that the average ping time is actually an overestimate of the WiFi network latency. This is because while measuring the ping time, there were many other devices contending for the WIFI. This would explain the 113 ms spike in the trial 6. Therefore, the actual round trip time would likely be closer to around 4 ms to 10 ms.

Figure 4: Example of Blue Screen

Model	Intel Core i5-8250U
Cores	4 x86 Intel Cores
Clock Rate	1.6 - 3.4 GHz
L1 Instruction Cache Size	32 KB
L1 Data Cache	32 KB
L2 Cache	256 KB
L3 Cache	6 MB
Line Size	64 Bytes

(a) Processor Specification

Model	Unknown
Size	8 GB
Bus	DDR3 SDRAM 1,866MHz
Width	64 bit

(b) Memory Specification

Model	M.2 Sata SSD 2280
IO Bus	PCIe
Type	SSD
Size	128 GB
Cache size	128 MB

(c) Disk Specification

Model	Killer Wireless-AC 1435
Protocol Specification	802.11a/b/g/n/ac
Speed	867 Mb/s

(d) Network Specification

Table 4: Dell XPS Specifications

7.2 Round Trip Time

Prediction: In this section we measure the round trip time of a packet getting sent from the client running on the Switch to a host server running on the Dell XPS and back. We predict that the remote round trip time will be the same as the average ping time. We also predict that the local round trip time will be much faster than the remote round trip time.

Methodology: For each trial, we created a C++ client on the Switch and a Python server on the Dell XPS. We connected the client to the server using the TCP library. Finally, we measured the total time it took for the client to send one character, the server to receive the character and send it back to the client, and the client to receive the reply from the server.

Trial #	Loopback Time (*ticks*)	μs	Remote Time (*ticks*)	μs
1	2830	147.4	79432	4137
2	2843	148.1	82224	4283
3	2911	151.6	81364	4238
4	4652	242.3	169351	8820
5	5549	289.0	74003	3854
6	7613	396.5	65376	3405
7	2815	146.6	58466	3045
8	2948	153.5	131139	6830
9	2959	154.1	73657	3836
10	6784	353.3	87749	4570
11	3113	162.1	66318	3454
12	2844	148.1	90526	4715
13	3256	169.6	100819	5251
14	3564	185.6	66197	3448
15	4095	213.3	68062	3545
16	3572	186.0	73817	3845
17	3125	162.8	85779	4468
18	3998	208.2	80462	4191
19	2856	148.8	70130	3653
20	3021	157.3	79357	4133

Results: Average Time for Loopback Round Trip Time: 3767.4 Ticks (196.2 μs)
Standard Deviation for Loopback Round Trip Time: 1344.0 Ticks (70 μs)
Average Time for Remote Round Trip Time: 84211.4 Ticks (4386 μs)
Standard Deviation for Remote Round Trip Time: 24798.3 Ticks (1292 μ)

Analysis: On average, the remote round trip time was around 4.386 ms, which was about one third of our original prediction. This is still consistent with our original prediction because the remote round trip time is within our estimated ping time recorded. The average local round trip time was about 0.1962 ms.

7.3 Peak Bandwidth

Prediction: In this section, we measure the peak bandwidth between the client running on the Switch to the host server running on the Dell XPS. We can calculate the maximum possible bandwidth using the formula $T \leq \frac{RWIN}{RTT}$, where T is the maximum throughput, $RWIN$ is the size of the TCP receive window (64 KB by default), and RTT is the round trip time [5]. Therefore, we calculate the bandwidth will be slightly less than $\frac{64 \text{ KB}}{0.0001962 \text{ s}} \approx 300$ MB/s for the loopback interface and $\frac{64 \text{ KB}}{0.001292 \text{ s}} \approx 50$ MB/s for the remote server.

Methodology: For each trial, we created a client and server through C++ sockets and measured the total time it took for the client to send 100,000 characters to the server.

Trial #	Loopback Bandwidth (GB/s)	Remote Bandwidth (GB/s)
1	0.155	0.00347
2	0.115	0.00377
3	0.127	0.00374
4	0.146	0.00431
5	0.106	0.00453
6	0.127	0.00685
7	0.129	0.00418
8	0.093	0.00279
9	0.080	0.00333
10	0.114	0.00198
11	0.120	0.00550
12	0.094	0.00459
13	0.134	0.00430
14	0.078	0.00627
15	0.149	0.00799
16	0.153	0.00358
17	0.153	0.00365
18	0.146	0.00221
19	0.120	0.00228
20	0.114	0.00296

Results: Average Time for Loopback Peak Bandwidth: 0.118 GB/s
Standard Deviation for Loopback Peak Bandwidth: 0.0233 GB/s
Average Time for Remote Peak Bandwidth: 0.00363 GB/s
Standard Deviation for Remote Round Peak Bandwidth: 0.00151 GB/s

Analysis: Our predictions are pretty far off the mark. On average, our peak bandwidth was 0.118 GB/s (118 MB/s) for the loopback interface and 0.00363 GB/s (3.63 MB/s). The discrepancy in the loopback bandwidth should be entirely due to how the Switch handles loopback requests in the *sockets* service. There could be inefficiencies in how the *sockets* service handles its requests that are exacerbated by sending large amounts of data.

For the remote interface, the discrepancy is much larger (our prediction is about 14x bigger than the measured bandwidth). There could be a few reasons for such a large discrepancy. The first is that the WIFI network we were using had many devices connected, which could have reduced the WIFI bandwidth to low levels. Similarly, as we see in the average ping time, the network latency was very inconsistent, and that could have had a larger impact when sending large amounts of data as opposed to the single byte we sent to measure round trip time. Second, similar to the loopback interface, there could be inefficiencies in how the *sockets* service handles large buffers of data to send and how the TCP/IP stack is implemented.

7.3.1 Connection Overhead

Loopback

Prediction: In this section, we measure the connection overhead of setting up and tearing down the connection between the client and server through the loopback interface. We predict that the setup time will be the same as the loopback round trip time, which is 3767.4 Ticks. since it does not need to go over the network. For teardown, we predict that it will take the same amount of time as the loopback round trip time because the client will send a FIN, receive and ACK from the server. We estimate that it will take about 3767.4 Ticks.

Methodology: In order to measure the connection overhead of setting up the connection between the client on the Switch and the server on the Dell XPS, we conducted 20 trials and measured the time for the accept call. To measure the tear down, we measured the time it took for the close connection call again on both the client and server.

Results: Average Time for Loopback Setup: 7180.7 Ticks (374.0 μs)
Standard Deviation for Loopback Setup: 3895.4 Ticks (202.8 μs)
Average Time for Loopback Teardown: 8785.3 Ticks (457.6 μs)
Standard Deviation for Loopback Teardown: 4179.0 Ticks (217.7 μs)

Trial #	Setup ($ticks$)	μs	Teardown ($ticks$)	μs
1	5583	290.8	6990	364.1
2	6796	354.0	7182	374.1
3	6647	346.2	7369	383.8
4	5650	294.3	6771	352.7
5	5847	304.5	8244	429.4
6	6491	338.1	11201	583.4
7	6005	312.8	26313	1370.5
8	5572	290.2	7943	413.7
9	7952	414.2	6938	361.4
10	6882	358.4	7474	389.3
11	6472	337.1	7861	409.4
12	6724	350.2	7081	368.8
13	6320	329.2	7249	377.6
14	23966	1248.2	8640	450.0
15	5970	310.9	7163	373.1
16	6345	330.5	10513	547.6
17	6961	362.6	8703	453.3
18	6156	320.6	7047	367.0
19	5543	288.7	7834	408.0
20	5732	298.5	7190	374.5

Analysis: From our measurements, the setup and teardown times were closer to double the round trip time. We believe the extra time for the set up was because additional data structures needed to be allocated when setting up a connection request, while the extra time for teardown was to deallocate the data structures when tearing down the connection.

Remote

Prediction: In this section, we measure the connection overhead of setting up and tearing down the connection between the client and server through the remote interface. For setting up, the client and server would perform a TCP 3-way handshake, so the client would need to send a SYN, receive a SYN-ACK, and send a ACK. Therefore, we predict that the setup time will be 1.5x longer than the remote round trip time. So, we estimate that it will take about 126317 Ticks. For teardown, similar to the loopback, we predict that the client will send a FIN and receive an ACK from the server. So, we expect it to be the same as the loopback teardown time, which was 8785.3 Ticks.

Methodology: In order to measure the connection overhead of setting up the connection, we conducted 20 trials and measured the time it took to complete the accept call on both the client and the server. Likewise, we measured the time it took for the client and server complete the close call. We performed the same experiments on both the remote and local connections.

Trial #	Setup ($ticks$)	ms	Teardown ($ticks$)	ms
1	124855	6.503	8382	0.436
2	116642	6.075	7258	0.378
3	183391	9.552	7183	0.374
4	137512	7.162	11080	0.577
5	160064	8.337	16407	0.855
6	555761	28.946	17979	0.936
7	179136	9.330	8622	0.449
8	284074	14.796	7293	0.380
9	150671	7.847	8230	0.429
10	139746	7.278	7480	0.390
11	136285	7.098	8277	0.431
12	485815	25.303	8199	0.427
13	137278	7.150	7331	0.382
14	241433	12.575	7103	0.370
15	221348	11.529	8139	0.424
16	165464	8.618	13386	0.697
17	136941	7.132	7388	0.385
18	131898	6.870	12335	0.642
19	166628	8.679	11355	0.591
20	132573	6.905	13273	0.691

Results: Average Time for Remote Setup: 199375.8 Ticks (10.384 ms)
Standard Deviation for Remote Setup: 115286.6 Ticks (6.005 ms)
Average Time for Remote Teardown: 9835.0 Ticks (0.512 ms)
Standard Deviation for Remote Teardown: 3190.6 Ticks (0.166 ms)

Analysis: From our measurements, the setup time was closer to double the round trip time. We believe the extra time for the set up was because of additional data structures needed to be allocated when setting up a connection request. The teardown time was accurate to our prediction, taking about the same amount of time as the loopback teardown time.

8 File System

8.0.1 File Cache Size

Prediction: In this section, we are measuring the file buffer cache size. We predict that this cache size would be fairly small given that 1. the existing operating system already constrains an application's memory usage on initialization and 2. the use cases of a large file buffer cache are fairly constrained when running Nintendo Switch games. We predict that the application will preload most of its assets into memory on game load, which would favor having a small file buffer cache size. For example, one of the only times that the file buffer cache will be used is during game saves.

Methodology: In order to determine the size of the file cache, we ran 256 trials of sequentially reading up to 256

4KB blocks from a 1 MB file. For each trial, we access the file once in order to ensure it is in the file cache and sequentially read up to the specified number of blocks. Then, we seek to the beginning of the file and measure the time it takes to read the file sequentially again according to the number of blocks. The measurements display the total read time divided by the number of blocks read. The size of the file cache will be represented as a spike in time it takes to reread the first block read from the file.

# of Blocks Read	Avg Time ($ticks$)	ms
1	5791	0.301
2	6376	0.332
4	4476	0.233
8	4546	0.237
16	7145	0.372
32	3472	0.181
64	3253	0.169
120	3380	0.176
121	3339	0.174
122	3310	0.172
123	3385	0.176
124	3229	0.168
125	3229	0.168
126	7021	0.366
127	6962	0.363
128	7134	0.372
129	6905	0.360
130	7148	0.372

Results: A graph of the results are shown in Figure 5.

Analysis: In order to highlight the file cache size, we chose a subset of the measurements for the results table to show the spike when 126 4KB blocks were read sequentially. However, the graph displays all the measurements. From the measurements, we see an increase in time at 126 blocks, which indicates that up to 126 blocks fit in the file cache at one time, which is 506KB. This is consistent with our original prediction that the file buffer cache size would be small, given that there is 4 GB of memory and only 506 KB are allocated to the file buffer cache.

8.0.2 File Read Time

Prediction: In this section, we measure the difference in time to access a file both sequentially and randomly. For sequential access, we predict that as the file sizes increase, the total read time should increase, but the average read time should be the same despite the file size. For random access, we predict that it will be larger than the sequential access time for the corresponding file sizes since there will be no prefetching.

Methodology: In order to determine the sequential read access time, we read different sized files that ranged from 1MB to 9MB. For each file, we averaged the time it took to sequentially read through the entire file in 4KB blocks. For the random read access time, using the same file sizes, we averaged the time it took to randomly access the same number of blocks as the sequential read access trials. To generate each random position within the file that will be read, we use the C++ random number generator to pick a random value and mod that value to be within the file size then seek to that position.

Results: Average Time for a Sequential File Read: 6033.0 Ticks (348.311 μs)
Standard Deviation for Sequential File Read: 1913.6 Ticks (11.536 μs)

# of Blocks Read	Sequential Avg Time ($ticks$)	μs
256	6808	354.6
512	6791	353.7
768	6545	340.9
1024	7226	376.4
1280	6460	336.5
1536	6655	346.6
1792	6491	338.1
2048	6590	343.2
2304	6621	344.8

# of Blocks Read	Random Avg Time ($ticks$)	μs
256	13618	709.3
512	16298	848.9
768	15634	814.3
1024	10699	557.2
1280	1800	93.8
1536	4910	255.7
1792	618	32.2
2048	2350	122.4
2304	24932	1298.5

Average Time for a Random Access File Read: 10095.444 Ticks (525.811 μs)
Standard Deviation for Random Access File Read: 7800.003 Ticks (406.243 μs)

A graph of the results are shown in Figure 6.

Analysis: As seen in the graph, the average read time for sequential read access was constant at about 6033 Ticks per block despite the varying file sizes. This result is consistent to our prediction. For the first 4 trials in random access experiments, we saw that the amount of time it took to randomly access file blocks was greater than the reported time for sequential reads. This can be attributed to the fact that these random file accesses do not take advantage of cache locality. However, for subsequent trials, we are unsure of the resulting improvements in performance of file random access, especially given the increase in file size. For small dips, we could potentially attribute these performance

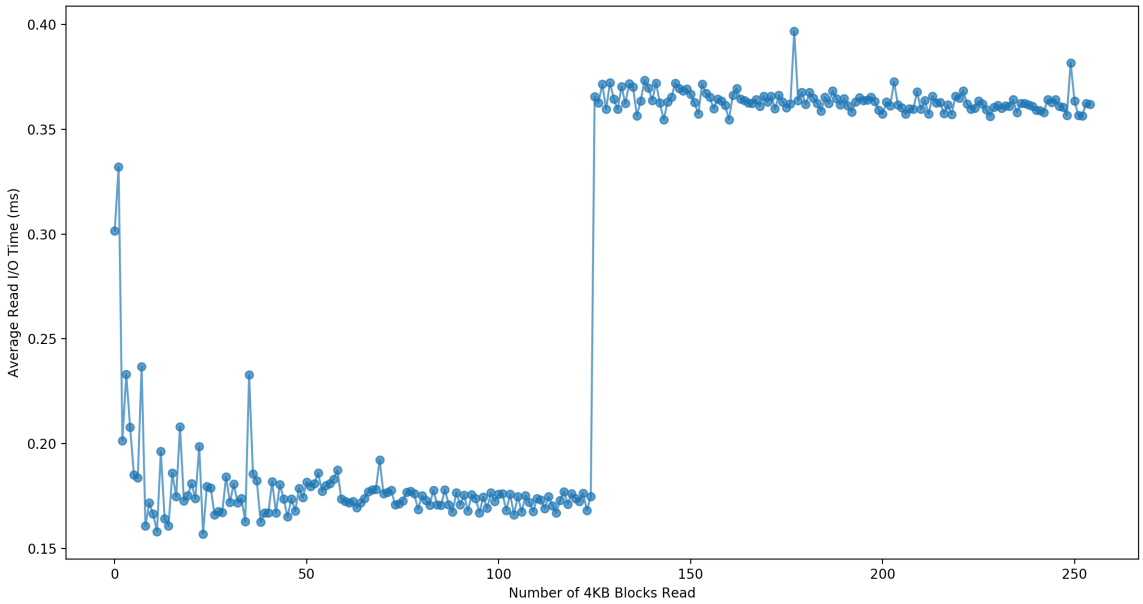

Figure 5: File Cache Size Results

speed-ups to more overall cache hits in the file buffer cache compared to prefetching done in sequential reads. We ran random access read multiple times but resulted in the same outcome.

8.0.3 Remote Read Time

The Nintendo Switch doesn't have any sort of built-in or created NFS utilities.

8.0.4 Contention

Prediction: We predict that with each additional thread, we expect the average read time to increase proportionally to the number of threads.

Methodology: In order to measure file read contention, we initially tried creating and starting multiple nonblocking background threads to randomly read from different files

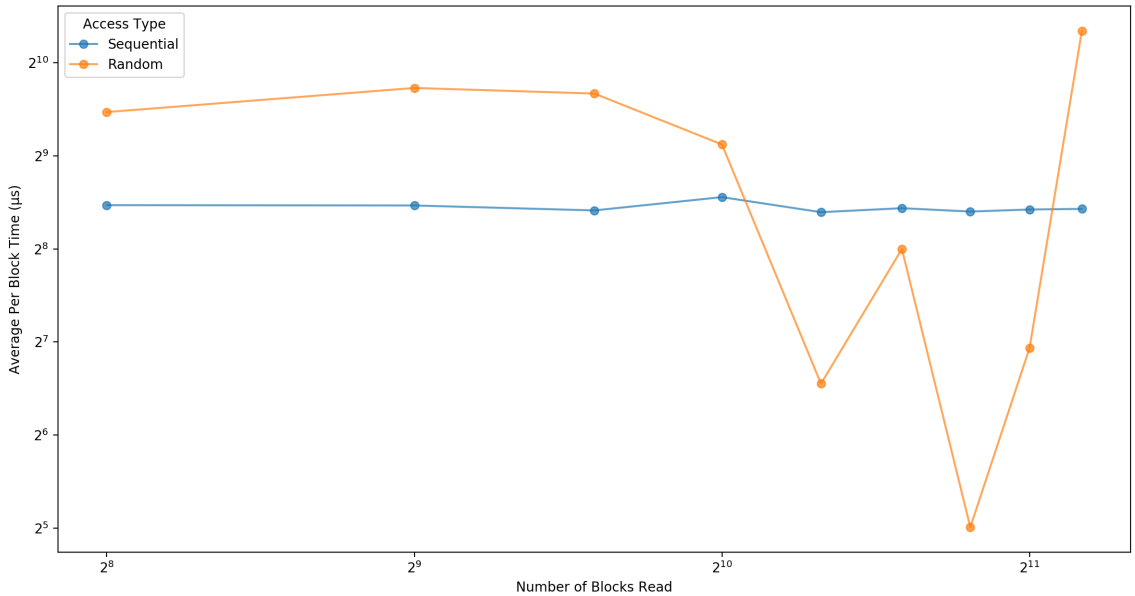

Figure 6: Random vs. Sequential Access Results

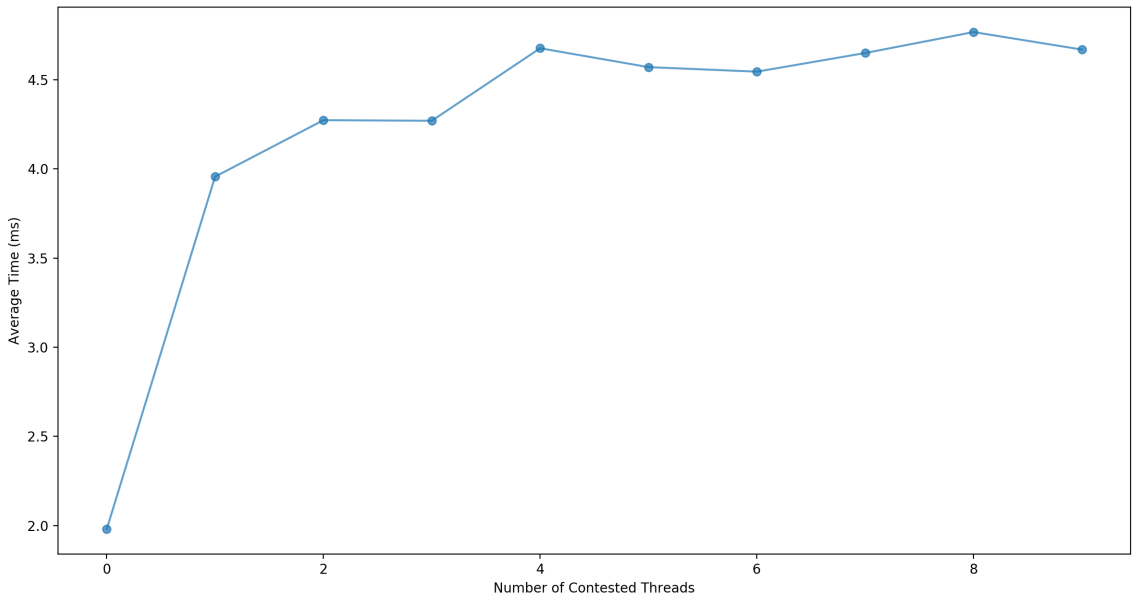

Figure 7: Average Time to Acquire Lock and Read 1 Block vs. Number of Contested Threads

and measure the time it took for the main thread to read 1 file block. However, this experiment continually resulted in blue screens, which suggests that the system could not support multithreaded file i/o. We also found online that a recent patch allowed an integrated file system service to support multithreaded file i/o. However this experiment also ended in similar results. Therefore, the remaining option that we had was to add a lock each of the threads trying to do file i/o and have the threads all contest for the same lock. For contending threads ranging from 0 to 9 threads, we ran 10 trials and averaged the time it took for the main thread to contend for the initialized file_read_lock and read 1 block from the file.

# of Contested Threads	Avg (*ticks*)	Avg (*ms*)	Std Dev (*ticks*)	Std Dev (*ms*)
0	38036	1.98	4279	0.222
1	75955	3.96	1958	0.102
2	82035	4.27	2948	0.154
3	81970	4.27	2047	0.107
4	89775	4.68	4635	0.241
5	87744	4.57	2924	0.152
6	87255	4.54	2587	0.135
7	89262	4.65	3434	0.179
8	91518	4.77	6117	0.319
9	89635	4.67	5391	0.281

Results: A graph of the results are shown in Figure 7.

Analysis: As the number of threads increased, the average time required for the main thread to contend for the initialized file read lock increased slightly. The average time the main thread took to read from a file with no contending threads doubled compared to the average time it took for the main thread to read a file block with one contending thread. This can be explained by the main thread waiting for that single thread to finish reading a file block before starting its own file block read. As the number of threads increased, we did not see a proportional increase in system ticks when multiple threads contented for the lock. This might suggest that the main thread could have likely always obtained the lock after a background thread had finished reading a system block. For further testing, we could have added a condition variable to measure the time it took for all other background threads to read before having the main thread read a file block, but we would likely see the same results as the thread context switch experiment. Thus given the limitations of the system, we are unable to fully test multithreaded file system i/o.

References

[1] Is my switch patched. "`https://ismyswitchpatched.com/`".

[2] Switchbrew. "`https://switchbrew.org/wiki/Main_Page`".

[3] The ultimate noob guide for hacking your nintendo switch. "`https://switch.homebrew.guide/`".

[4] Vulnerability disclosure: Fusée gelée.

[5] Measuring network throughput. "`https://en.wikipedia.org/wiki/Measuring_network_throughput`", Oct 2019.

[6] Larry McVoy and Carl Staelin. Imbench: Portable tools for performance analysis. *USENIX*, Jan 1996.

[7] Gauvain Tanguy Henri Gabriel Isidore Roussel-Tarbouriech, Noel Menard, Tyler True, Tini Vi, and Reisyukaku. Methodically defeating nintendo switch security, 2019.