1 INTRODUCTION

1.1 Motivation

Datacenter applications must be scheduled to provide \( \mu \)-scale tail latencies and high throughputs even for highly dispersive workloads [10]. Modern data plane operating systems support these requirements by spin-polling the NIC on dedicated CPU cores and skipping the bloated kernel network stack with kernel-bypass networking [6, 8–11]. However, many of these systems just pin threads to cores and let CFS do the scheduling [6, 8, 9, 11]. This approach has two downsides. The first is that this allocates for peak load, which wastes hardware resources. The second is that CFS was designed for the millisecond scale, not the microsecond scale, so there is no way to reserve ms-scale applications with low-priority background work that gets good performance and is responsive to bursts in network traffic.

Shenango fixes this problem. Shenango has a built-in IOKernel that supports fine-grained core reallocations between applications (up to every 5 \( \mu \)s) and uses a novel algorithm to predict which applications are "falling behind" in handling their own network traffic. Shenango can quickly (i.e., every 5 \( \mu \)s) allocate additional cores to applications that are falling behind, whether due to a burst in network traffic or a batch of long requests that have arrived, allowing the application to catch up. Additionally, Shenango allows low-priority batch applications to run and can quickly context switch to latency-sensitive applications as work arrives, which is not possible to do in systems that pin threads to cores with CFS. Shenango can linearly trade off low-priority batch application throughput for high-priority low-latency application throughput (on the \( \mu \)-scale) while maintaining comparable tail latencies to other low-latency data plane operating systems. In other words, Shenango performs just as well as these other systems but has much greater CPU efficiency.

1.2 Original Results

The Shenango authors showed that Shenango was able to match Zygos [11], Arachne [12], and vanilla Linux with higher CPU efficiencies for the reasons described above. First, the Shenango authors showed that by supporting \( \mu \)-scale core reallocations, they were able to beat an optimal ms-scale core allocator (Figure 1 of the paper). This was an important baseline result to show that there is a need for more fine-grained core allocators for applications that have tail latency sensitivity on the order of microsseconds. The paper then goes on to compare against Zygos, Arachne, and vanilla Linux for a key-value store (KVS) workload with memcached [3]. The authors demonstrated that Shenango was able to achieve comparable (or better! Especially in the case of vanilla Linux) 99.9th percentile median and tail latencies to these systems while still offering much higher batch application throughputs (Figure 3 of the paper). Essentially, they were able to give unused cycles from the latency-sensitive workload to the batch processing workload, which these other systems are unable to do since they either allocate for the peak or were not designed for this time-scale.

Figure 4 of the paper shows that Shenango is able maintain low 99.9th percentile latencies across a variety of service time distributions (with a mean of 10 \( \mu \)s), while still supporting batch processing. These distributions are the constant distribution, the exponential distribution, and the bimodal distribution. Zygos has comparable tail latencies to Shenango for all three service time distributions, but Zygos is not able to provide CPU time to the low-priority batch applications whereas Shenango does. Arachne and Linux have poor tail latency performance and while they provide more batch application throughput than Zygos, their batch application throughput still falls short of what Shenango provides.

Figure 5 shows that sudden spikes in network load require quick core reallocations; Shenango is able to offer much lower and stabler 99.9th percentile latencies for bursty workloads (i.e., the tail latency doesn’t jump momentarily even though Shenango needs to provide more cores to the network application) while Arachne (with its slower core reallocation speed) experiences huge spikes in latency with even modest spikes in load.

We reproduced Figures 1, 3, 4, and 5. That being said, Figure 8 in the Shenango paper highlights that with larger core allocation intervals, workload tail latency significantly degrades. Shenango’s 5 \( \mu \)s core reallocation interval is essential to its good performance.

1.3 Reproduction Methodology

We reproduced the following figures:

1. Which compares an optimal 1ms core allocator with Shenango.

2. Which all plot 99.9th percentile (and median for figure 3) latencies as well as batch operation throughput for Shenango with a variety of workloads.

Our goal is to verify the central claims made in the Shenango paper using the hardware and network available to us at

Shenango Reproduction Project: Final Report
Nikhil Athreya, Jack Humphries
Stanford. The Shenango paper compares how workloads run on Shenango with how they run on ZygOS, Arachne, and vanilla Linux, but due to time constraints, we have chosen to reproduce only the Shenango results rather than also include the other systems’ results.

We first reproduced Figure 1, in which we simulated an optimal 1ms core allocator and plotted its CPU efficiency on a minimum number cores vs. throughput with the constraint that the 99.9th percentile tail latency had to be at most 100μs.

We next set up Shenango on Stanford machines, which required changing some of the code provided by the authors. After setting up Shenango, we ran the remaining workloads for Figures 3, 4, and 5, plotted the latencies and throughputs, and compared these against what the paper presents. See Section 3 for more details, including hardware, OS, and network setup.

1.4 High-level Results

We found that our implementation of a 1ms optimal core allocator (for Figure 1 of the paper) closely matches what the paper describes. Our optimal core allocator seems to have worse performance than a Shenango core allocator with a 5μs interval, as expected (note that this is expected because the optimal core allocator with a 1ms reallocation interval has a higher interval than Shenango’s, which is 5 μs).

Our results for Figures 3, 4, and 5 closely match the results in the paper, after accounting for differences in our experimental setup. Our results for Figure 3 show that Shenango can linearly trade off low-priority batch application throughput for low-latency application throughput. Our results for Figure 4 show that Shenango can handle a variety of request time distributions while maintaining low latency and maintaining the trade off with batch application throughput. Since Shenango can handle a wide variety of request time distributions and get good performance, this indicates that Shenango is good at figuring out which low-latency applications are “falling behind” and giving them more CPU cores so they can catch up. Lastly, Figure 5 shows that Shenango maintains good stable performance even when network traffic spikes, again indicating that Shenango is good at quickly allocating cores to low-latency applications as needed.

2 PRIOR WORK

The Shenango IOKernel allocates applications with a minimal number of cores and reallocates the cores every 5 μs as needed. Shenango uses kernel-bypass networking provided by DPDK [1] to bypass the bloated Linux networking stack and interact with the NIC directly. Both μs-scale scheduling and kernel-bypass networking are areas of active research and Shenango draws (and improves upon) much prior work.

IX [6] was a major contribution to these areas of research. IX is a data plane operating system that combines kernel-bypass filtering (provided by DPDK) with hardware virtualization (provided by Dune [5]) to break the “tradeoff” between low latency, high throughput, protection, and resource efficiency. By using kernel-bypass to access NIC queues directly and skipping the bloated kernel stack, IX was able to get low latency (on the μs-scale) and high throughput. By using hardware virtualization via Dune (which is accelerated by Intel VT-x), IX was able to run its data plane component in ring 0 of non-root mode and isolate network applications in ring 3 of non-root mode, to protect the data plane from the applications. Resources freed up by IX can be used for other applications or put to sleep. IX uses RSS to distribute packets to cores, and pins a thread to each core. Thus, IX is the data plane and the Linux kernel is the control plane.

The problem with IX is that highly dispersive workloads handled by IX have poor tail latencies as IX is not work-conserving (due to its use of RSS) or preemptive (so short requests get stuck behind long requests). ZygOS [11], which builds off of IX [6], fixes the work conservation problem. ZygOS still uses RSS and still pins a thread to each core. However, ZygOS implements work-stealing to help improve work conservation, which in turn improves performance.

Arachne [12] is a thread and core management system that can reallocate cores between applications at a fine granularity. Arachne was built to support applications whose threads start, do work very quickly, and then exit.

Other related work includes Shinjuku [9], which came out at NSDI 2019 alongside Shenango. Shinjuku is also based off of IX, but it fixes the work conversation and avoids head-of-line blocking. Shinjuku maintains a central dispatcher that the NIC sends all packets to (RSS is not used). The dispatcher then assigns work to cores based on which cores are idle, which makes Shinjuku work-conserving. Shinjuku also preempts cores that are taking too long to process a request using highly optimized posted interrupts, so Shinjuku avoids head-of-line blocking as short requests are not stuck behind long requests.

Lastly, Mind the Gap [8] explores offloading the Shinjuku dispatcher to a smartNIC, which is a good idea as the smartNIC is a centralized location in the system that has global visibility of all packets entering and leaving the system. Additionally, the smartNIC can implement several optimizations such as steering packets to certain NUMA nodes and using DDIO to get better cache performance in the host, which are optimizations not possible in vanilla Shinjuku as the dispatcher runs on the host CPU. The prototype system in the paper gets better performance in some cases than Shinjuku as it doesn’t waste a core on the host CPU for scheduling, but it also shows the need to put a P4 pipeline (or something similar) in smartNICs in order to scale as well as when the
Shinjuku dispatcher runs on the host Intel CPU. The smart-NIC used in the paper contained slow ARM cores and had high NIC-host communication overhead.

Like the systems above, Shenango uses DPDK to do kernel-bypass networking. On the scheduling side, Shenango uses a concept known as two-level scheduling [10], which first uses a spatial-scheduler to allocate cores to applications and then uses another scheduler on top of this one to allocate threads running on cores. This concept, while not a new one [14], is a very unique approach to scheduling since it decouples core allocation from thread scheduling. Other systems, such as Akaros [13] and Callisto [7], use the approach of two-level scheduling, but Shenango innovates on top of this by also considering the NIC queues in its first-level scheduler [10].

3 DESIGN AND METHODOLOGY

3.1 Shenango’s Design

Shenango optimizes CPU efficiency by allocating each application the least number of cores as possible such that adding an extra core to the application only decreases its delay by at most a couple of microseconds (in other words, we see most of the performance gains without wasting a whole core for something like a 1% decrease in latency). If an application is allocated too few cores it is called “compute congested,” since it could see significant gains in performance by allocating more cores. Shenango uses the spare cores for low-priority batch processing, and Shenango can quickly reallocate cores every 5 μs. Shenango’s two key contributions are:

1. A congestion detection algorithm, which can decide if an application is compute congested and could benefit from additional cores.
2. An IOKernel, which runs on its own core, that gives the congestion detection algorithm information about packet queues for each application. It provides applications low-latency access to network packets since it is a layer of abstraction on top of NIC hardware queues that applications can build off. Additionally, the IOKernel can also efficiently re-route packets intended for one core to another, which means that Shenango can have a very short core allocation interval (5 μs).

Applications each have their own runtime where they have access to multiple kernel threads (up to the number of threads allocated for the application) that their user threads can be multiplexed on top of; user-level threads are placed into queues, so they can be moved between cores to facilitate work-stealing.

Logically, each application (with a guaranteed and a burstable number of cores) sits on top of the IOKernel (which is also running on its own core). The IOKernel facilitates communication between applications and the OS kernel by allowing applications to link to the Shenango runtime (which is a library). Additionally, the IOKernel has direct access to the NIC’s queues via kernel-bypass networking, enabled by DPDK.

3.2 Reproduction Methodology

We first began by reproducing Figure 1 of the paper by simulating an optimal 1ms core allocator on a workload with a mean service time of 10μs per request and a 99.9th percentile tail latency hard limit of 100μs at various request throughputs to determine the minimum number of cores required to satisfy the constraints. We accomplished this by simulating 1μs-per-tick CPU(s), and re-allocating cores every 1000 ticks. We did this across multiple throughputs, ending up with our reproduction of Figure 1.

Simultaneously, we also worked on acquiring the necessary hardware to run Shenango. We tried to match the specifications from the paper (mentioned in their evaluation section) as closely as possible, since any machine hardware or software differences could worsen the quality of our reproduction. After acquiring the hardware, we set up Shenango and any relevant experiments. Fortunately, both Shenango [2] and its benchmarks [4] were open source, which meant that we could run the exact same experiments used in the paper.

Getting Shenango up and running was a non-trivial task for us. It required debugging a lot of networking issues that couldn’t be debugged with Linux commands due to the IOKernel’s use of DPDK, which takes over NICs from Linux. There was a lot of trial and error involved when tweaking Shenango’s configuration parameters; in fact, we initially started setting up Shenango on a less powerful machine and it turned out we actually needed a machine with a better NIC to be able to properly run the system. While there was an overhead to getting the system up and running, we do appreciate the learning experience this entire process provided. On that note, we would especially like to thank the Shenango authors for not only open-sourcing their system and its benchmarks, but also for being very timely in their responses to our (many) questions.

4 REPRODUCTION RESULTS

4.1 Experimental Setup

For our reproduction, we used one server and one client, both running on separate machines. The machines are both equipped with 10G 82599ES Intel NICs and are connected by a 10G switch.

The server is equipped with an Intel Xeon E5-2630 CPU with 2 NUMA nodes. Each node has 6 physical cores, with 2 hyperthreads per core. Thus, there is a total of 24 hyperthreads, with each hyperthread was running at 2.3 GHz. The
server has 64GB of RAM. The operating system is Ubuntu 18.04 running on Linux kernel version 4.15.0.

The client machine has identical specs as the server.

We downloaded the Shenango source code along with the experiment source code from GitHub. We reconfigured the experiment script to work with our network (e.g., we had to change host names and IP addresses). We manually configured about 16 GB of hugepages on each machine and bound the NICs to DPDK drivers as needed.

The script took care of other configuration details. For example, the script used numactl to restrict the Shenango programs from only running on NUMA node 0 and only allocating memory from NUMA node 0. Additionally, the script opens /dev/cpu_dma_latency, writes 0 to it, and keeps the file descriptor open to prevent the CPU from entering deep C-states.

We limited each Shenango application in the experiments for Figures 3, 4, and 5 to 4 threads each in order to ensure one client machine could saturate the Shenango application.

4.2 Figure 1

Our reproduction of Figure 1 pretty closely matches the original figure from the paper. We plotted our simulated 1ms optimal core allocator against the reference Shenango curve with a 5μs core allocation interval. Shenango is still often able to outperform the more coarse-granular optimal core allocator, which is what the figure in the original paper showed as well.

One notable difference between our reproduction and the one from the paper is that our 1ms optimal core allocator seems to perform a bit better than the one shown in the paper. As an example, we can see that at 2 cores, our simulated allocator is almost able to reach the performance of Shenango at its peak, while in the paper, it is still far beneath Shenango in terms of efficiency.

This does not diminish the benefits of Shenango as shown in the paper, since the point is that Shenango is able to outperform a more coarse-grained optimal core allocator when considering a wide range of input rates (not just the peak input rate for a given number of cores while maintaining a maximum 99.9th percentile tail latency). We can see that if we continued to plot input rates beyond what was shown in the figure, giving the coarse-grained optimal allocator more and more cores, Shenango would probably start falling off, since its efficiency seems to flatten around 87.5%. This would happen more quickly in our simulation, since the efficiencies for our simulated allocator grow at a faster rate than for the allocator in the paper.

4.3 Figure 3

Our Figure 3 is similar in shape to the Figure 3 in the Shenango paper. Our graphs saturate at a lower memcached throughput because we use a smaller number of cores on the server. Additionally, we have a lower batch throughput, particularly when the memcached load is 0 requests per second, because we use a small number of cores on the server. However, the shapes of our graphs are similar to the shapes of the graphs in the Shenango paper, which is ultimately what matters. Additionally, the third graph in our Figure 3 shows that batch throughput decreases linearly as memcached load increases, which confirms the claims of the Shenango paper (i.e., batch throughput can be linearly traded off with low-latency application throughput).

4.4 Figure 4

Our Figure 4 has similar shapes to the Figure 4 in the Shenango paper for all three request time distributions (constant, exponential, and bimodal). Again, the actual numbers of the x-axes and y-axes differ from the paper due to our different configuration, but the shapes are the same. This confirms that Shenango can handle dispersive workloads and get good
Figure 3 from the original Shenango paper.

Our reproduction of Figure 3.

Figure 4 from the original Shenango paper.

Our reproduction of Figure 4 (constant request service time distribution).

tail latency for the low-latency application while still allowing the low-priority batch application to run. Shenango can handle these highly dispersive workloads and still let the low-priority application run because its IOKernel can allocate cores very quickly (every 5 μs) to the low-latency application when it is falling behind, while otherwise letting the low-priority batch application use the cores.

4.5 Figure 5

Our Figure 5 is similar in shape to the Figure 5 from the paper. Again, the actual numbers of the x-axes and y-axes differ from the paper due to our different configuration, but
the shapes are the same. Even when the throughput changes very quickly, Shenango is able to handle the burst without having its tail latency explode temporarily. This confirms the claim of the paper that the IOKernel can quickly (every $5 \mu s$) allocate cores to an application that is “falling behind.”

5 LIMITATIONS
While our reproduction is a good first-order approximation of the results in the paper, there were still a couple of limitations in our evaluation that prevented us from fully reproducing the results from the paper.

Setup of other systems: Due to time-constraints, for our evaluation, we just aimed to reproduce the optimal 1ms core allocator for Figure 1 and the Shenango curves for Figures 3, 4, and 5. However, some of these figures also had other systems (like Arachne, Linux, and ZygOS) with which Shenango was compared against. It would have been useful to benchmark these systems on our experimental setup as well to be able to contextualize the Shenango results. That being said, the Shenango results we reproduced verify the central claims of the paper.

Figure 5 from the original Shenango paper.

Our configuration: Our server and client configurations were different as we did not have access to the same hardware as the authors, mainly because the equivalent hardware at Stanford was in use by other researchers. It would have been helpful if we could reproduce Figures 3, 4, and 5 on the same hardware.
Our reproduction of Figure 5.

Having a different software configuration could also be an issue. Some possible differences are described below:

(1) Different kernel configuration parameters used to compile the kernel (even though we used a similar kernel version to the one mentioned in the paper).
(2) Different compiler versions (which could potentially have resulted in vastly different performances due to regressions in the compiler’s optimizer). This is not as unlikely of a problem as it initially seems, since Shenango uses Rust for some of its experiments, which is still a new language with a fast-moving compiler.
(3) Different library versions for externally linked libraries (using newer versions of software could have resulted in us benefiting from crucial performance fixes).

While individually, each of these may seem negligible, when combined, they could have impacted our measured performance in a non-negligible way. The issue here is that since the original paper doesn’t mention many of these configuration choices (they just mention general kernel/OS versions as well as the specifications of the machines they used), it would be very difficult for us to exactly reproduce what they did. Although, the argument could also be made that the system should be generalizable so these differences shouldn’t matter, our goal was to reproduce Shenango, which means we would want to keep all of these differences to a minimum and see whether the results claimed in the paper hold.

Incomplete reproduction of all figures: The figures we reproduced, mostly show the end-to-end performance advantages offered by Shenango. However, reproducing some of the other figures in the paper (6, 7, and 8) would have been valuable as well. Since we did not get a chance to reproduce all of the figures in the paper, our reproduction is fundamentally incomplete.

Figure 6 dissects how much time at a microsecond scale is spent by Shenango in: the IOKernel (which decides how many and which cores are assigned to an application as well as handling network I/O) and wakeup and preemption (Shenango uses Linux syscalls to switch between kernel threads, which adds overhead). They found that their system only added a couple of microseconds more overhead over the traditional DPDK approach to kernel-bypass networking. While we may not have been able to reproduce the exact times stated in the paper due to differing hardware and/or software configurations, at least confirming that our numbers are in the same vicinity would have been helpful.

Figure 7 looks at how well a design choice they made in their system (their work-stealing implementation) performed. They found that they were able to scale really well, maintaining similar tail (99.9th percentile) latencies compared to their ZygoOS peer, which suffers from a smaller number of connections due to RSS (receiver-side scaling) distributing flows unevenly across cores (which should be more apparent with a smaller number of flows). In addition to reproducing the figure itself, it would have also been interesting to look at how Shenango performed with and without its work-stealing implementation; looking at how much this choice affected the overall performance would have given valuable insights into one of the key design choices of this system.

Finally, Figure 8 shows that increasing core allocation intervals for Shenango degrades performance. This would have been a nice complement to our reproduction of Figure 1, since we could potentially compare how Shenango with 1ms core allocation intervals compares to (1) an optimal 1ms core allocator and (2) Shenango with a smaller core allocation interval (e.g., 5μs as was used in the paper). Having all three curves reproduced would have provided valuable context for the performance of the system.

6 FUTURE WORK

If we had more time, there are more things we would have liked to do.

Scheduling across layers: Does it make sense to move aspects of Shenango scheduling onto a smartNIC or a switch?
How should existing scheduling policies running on smart-NICs and switches interact with Shenango, particularly as CXL comes out? Is there an easier way to write a coherent scheduler that spans multiple layers?

**Varying core allocation intervals:** One experiment that would have been interesting to do is to compare how close to optimal Shenango was. Figure 1 from the original paper compares Shenango with a 5μs interval between core allocations against an optimal 1ms core allocator, but it would have been valuable to see how far away Shenango was from an optimal 5μs core allocator, due to packet processing and other overheads.

**Detailed end-to-end traces:** Figure 6 from the Shenango paper gave a detailed trace of time spent in Shenango’s IOKernel, but it would have also been useful to see how much time Shenango spent on each step required to process a single request (say with a mean service time of 10μs as Figure 1 from the paper) from end-to-end. This would have been a valuable to understand, since with a 10μs mean service time, Figure 6 shows that Shenango spent almost 15μs just in the IOKernel, which would suggest there is non-negligible overhead in networking present. Additionally, having end-to-end traces, like what was mentioned above, for all of the systems in the paper would have been useful, since we could then have compared Shenango to other systems in terms of relative performance for different tasks like: time spent in the networking stack, time spent doing useful work (i.e. actually processing the request), how much overhead there was in each system, etc. Once we have this information, we know what to optimize next. However, gathering this information at the microsecond-scale is challenging.

**REFERENCES**