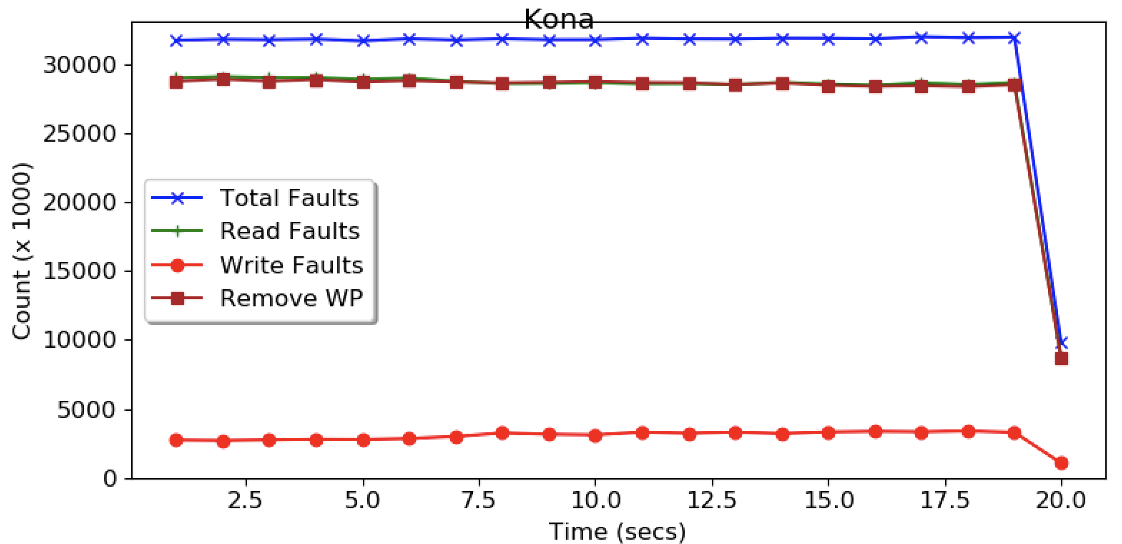
Journal

# Week 12

## 09/08

* Even with 0.2% WRITE workload, it seems like memcached is still dirtying all the pages no matter the workload type. As I run Kona in CONFIG\_WP mode, most page faults do happen on a load but the application immediately comes back to remove the write protection as shown by equal number of read faults and remove WP operations.



It’d be interesting to know what in memcached causes this.

* Do we want to look into scaling Kona with green threads/co-design with scheduler, etc? Not really. We want to get out of Kona bottlenecks with minimal effort (and this goes for page fault related challenges in general) because all these will disappear when CXL shows up, so we want to be tackling problems that come with/persist even in the CXL setting. We want to use Kona as a platform to tackle those problems since CXL is still not here, but we don’t want to spend too much time improving it.
* To minimize page faults, first try these:
  + Understand what’s causing so much dirtying in memcached and see if we can easily fix it
  + Use a different distribution like Zipf workload, not uniform random
  + Look at how madvise is issued right now, can we batch them?
  + Perhaps after exploring these, we can move to Nadav’s patches.

## 

## 09/06

* Finishing up slides for final presentation 09/07
* Talked to Nadav. **TODO** Add notes/takeaways
  + Madvise cost can be cut down by issuing a batched call for multiple notifications using process\_madvise. However, this operation does not yet support DONT\_NEED advice, which is what eviction uses to remove the virtual address of evicted pages. This is what Nadav was gonna add soon?
  + Using io\_uring would perform the uffd copy operation from the faulting thread instead of the handler thread, improving the throughput of the handler thread and removing it as a bottleneck. Although, right now, the eviction thread is the bottleneck.
* Thoughts on whether we can use green threads to improve cpu efficiency and scalability of remote memory handlers
  + Come up with a longer-term solution: assuming we have Nadav’s optimizations, some overheads still remain. How can green threads help minimize these overheads?
  + Co-designing scheduler and remote memory handlers.
* With only 0.2% writes, how is it that we are seeing the number of evictions (network writes) similar to the total number of page faults? Since most of the pages won’t be dirty, shouldn’t they just be dropped instead of writing back, with the CONFIG\_WP kona setting?
  + TODO: Investigate
  + Is it possible that even with GETs, memcached is dirtying the memory?
* Plan for writing up this work in the next week

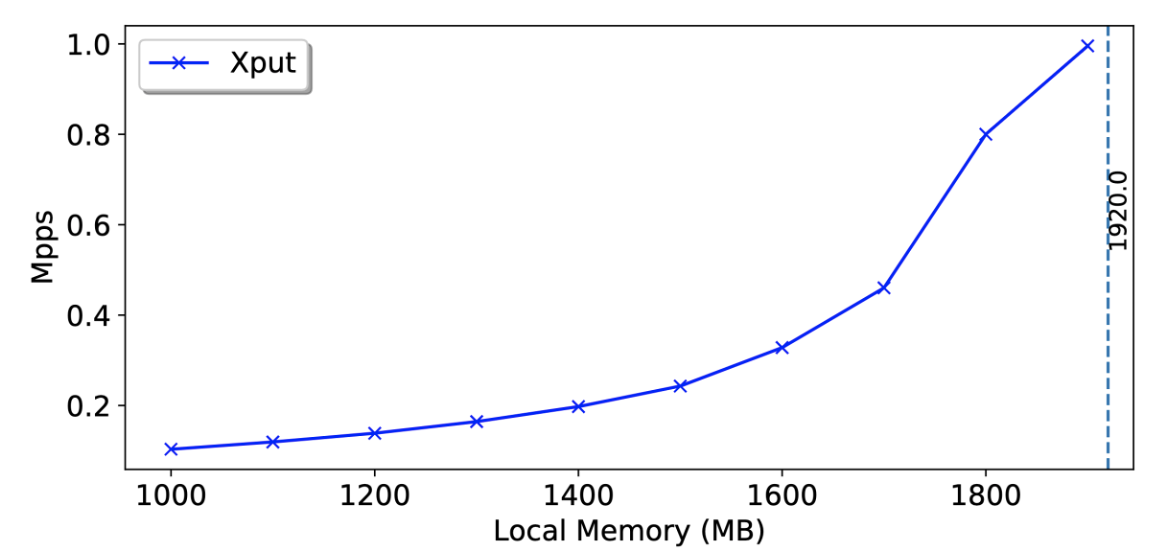
# Week 10 & 11

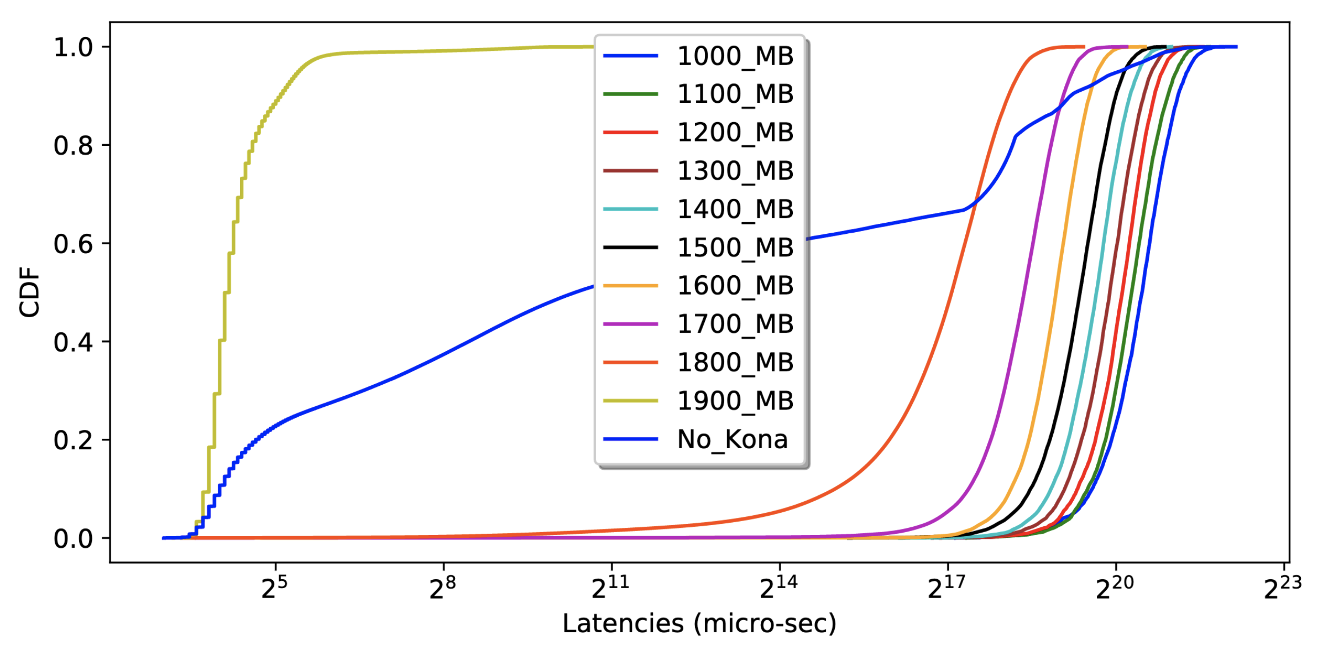
* Missed notes in both these weeks. **TODO** write summary/highlights
* **TODO** Include the charts with takeaways

# Week 9

## 08/20

## 08/19

* **Racklette Weekly Notes**
  + What would the rack-level scheduling look like? Would it be a mix of coarse-grained global scheduling combined with node-level fine-grained scheduling?
  + [**Marcos**] Shenango decouples core allocation from the thread creation. Perhaps we should extend this principle to the rack-scale as well. On that note, can we move the current core allocation decisions and move it to the controller and see what the overhead looks like? This can be the V1. V2 would be the actual scheduling that needs migration capability for the threads.
    - [**Anil**] Check on what timescales Shenango is making its current core allocation decisions. TODO
  + We can try and use core allocation to offset the blocking of threads using page faults.
    - We need to see what the overhead of such a control operation is compared to the actual page fault servicing time? Would it be much worse than simple properly-designed upcalls?
    - This faces a challenge though: Presumably on a page fault, we can spawn a new kernel thread that occupies the abandoned core while the page fault is being serviced. But when the page fault returns, there’d be two kernel threads now active on the core which brings time-slicing and kernel scheduler into the picture. We also just cannot discard the first kernel thread because it has the state of the user thread that is blocked, which cannot be retrieved without it running again. Section 2.2 of [SchedAct](https://web.eecs.umich.edu/~mosharaf/Readings/Scheduler-Activations.pdf) discusses this.
    - Upcalls are implemented in Bespin (but in Rust?). We should discuss the upcalls and their complexity next time. Read the Lithe paper.
  + [**Marcos**] We should however think beyond page faults and should not base our design too much around them; because with CXL and hardware-based remote memory, page faults won’t be an issue.
* **Meeting with Radhika Notes**: Analyzing initial results with Kona and questions to be answered:



* + Why is the baseline throughput so low on 12 server cores? See how Shenango scales with increasing cores, and also compare with the Shenango paper.
  + Why is memcached without Kona doing much worse than Kona with 100% local memory?
  + Get the latency plot in a time-series fashion. Maybe we can see some patterns in pagefaults.
  + See if we can get some stats on pagefaults: count, duration of each fault, etc. Maybe we can estimate the about of CPU lost to blocking page faults using this metric.
  + The network question is still open: How are DPDK and RDMA working together?
* How can we transition from Shenango to RThreads and co-scheduling plans? Think about a concrete plan. Is Shenango the right model for RThreads? Or would frameworks designed for parallel programs like Lithe or Callisto a better fit? Depends on which applications we plan to run on our rack.

# Week 8

## 08/12-18

* Kona integrated in both code and scripts. Got initial numbers on throughout by varying local memory. Focusing on the results now.

## 08/11

* Replaced **malloc**()s in Memcached with **rmalloc**()s: Time to get the traffic flowing!
  + Looking at traffic generator client to slow it down for debugging
* What bottlenecks do you see in a (Shenango + Kona) system?

## 08/10

* Resolved the pagemap access issue. Tricked memcached into working by changing just the effective user. So, with user ids, there's real vs effective: real is what you are, effective is what kernel thinks you are (which can be some other user if you have permission to be that user). Memcached was changing real user id to ayelam, hence losing root privileges, but it looks like it keeps working even if i just change the effective user id (I don't know why memcached needs to run as another user but at least it seems satisfied with the effective user). By changing just the effective user, you can switch back to root whenever you like - so in shenango when I access pagemap, I'm temporarily switching effective uid to root and switching it back when it's done.
* Kona initializes now. Isolated Shenango’s Iokernel and runtime cores from Kona’s worker cores.

## 08/09

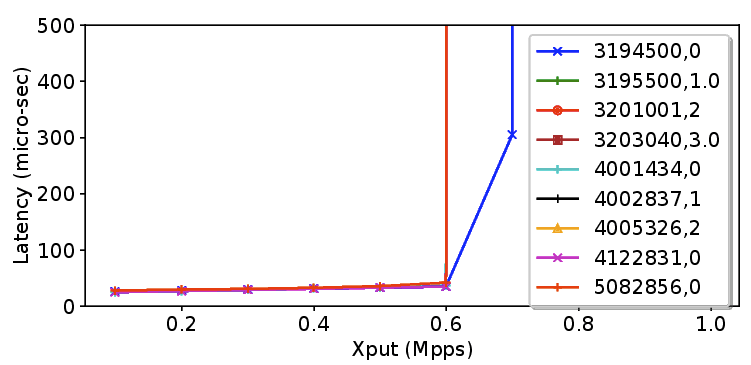
* Sudo with -u flag seems to solve the memcached root privileges issue.
  + But it introduced another issue: Shenango accesses proc resources like pagemap which are only available to the user that created the process; but memcached changes the process uid/gid to the non-root user specified by -u flag that doesn’t have access to these resources. Exploring capabilities to solve this issue.
* Spent all day on setting the right capability (CAP\_SYS\_ADMIN) for the process to get pagemap access with non-root user. Still not there yet.

# Week 7/12

## 08/06

* Pankaj Mehra talk (Data, a memory centric view)
  + The cost of memory is the biggest challenge in scaling data centers; Memory is virtualized at server level, but not at the data center level
  + DRAM cost trends are not going down. Cites zSwap from Google. Ray and memory disaggregation are important to keep the TCO down.
  + CXL is still some time away in the future; can’t depend on it? On that note, I need to learn more about CXL; I’ve been hearing about it a lot recently.
* Integrated Kona with memcached at compile time. [Build](https://gitlab.eng.vmware.com/ayelam/rmem-scheduler/-/commit/9dcb3547afe26b30fe07b084367ffe93d010d584) works!
  + Had to export a kona library and insert it into memcached makefiles
  + Using pbmem mode, where I’m going to initialize Kona from memcached using *rinit()* and use rAPIs (rmalloc to allocate memcached memory)
  + Kona initializes from within memcached! (registered itself with the rack controller and memory server)
    - **Issue**: Kona needs sudo but memcached seems to dislike it. Tried suppressing the errors but ended up in seg fault :( **Maybe start with a simple Shenango benchmark first?**
    - Would shenango runtime be ok with sudo? Normally we shouldn’t be needing it. Is there a conceptual reason why the Kona needs to run as root, or was it a technical simplification?

## 08/05

* Memcached baseline numbers of the new Kernel
* 
  + Not sure why it is maxing out at 0.6 Mpps. Different lines correspond to different runs, whereas the second number is the starting offered load in Mpps.
  + Need to change the request distribution and other settings. Where is the bottleneck coming from? Can we start slow in the client?
  + I need to understand the client better; it's written in Rust.
* Shenango + Kona challenges
  + Start with the build part first

## 08/04

* Look at Kona application
  + Ask Jiacheng for any Kona bugs he found to preemptively fix them and save some time.
* **TODO** What are asynchronous page faults in KVM? Can we use them as scheduler activations?
* Shoot an email to Nadav ✅
* Having trouble understanding how Shenango’s dpdk net stack interacts with the regular net stack on other machines. Currently it can only interact with Shenango-based netstacks on other machines, although supposedly it should be able to interoperate.

## 08/03

* Kona integration
  + Q: Which mallocs should I interpose? Clearly all the memcached ones, but how about mallocs in the runtime where runtime stores user thread metadata, etc? There’s only one malloc I think in memcached, maybe I should just interpose that one.
  + TODO Look at Jiacheng’s ML application and how he integrated his memory APIs for example.
* Got new numbers for memcached on new kernel (post chart here **TODO**)
* Also modified Shenango to run on a custom numa node instead of hardcoded socket 0. Our NICs are on socket 1.

## 08/02

* Figured out the bug; Shenango now works on the latest kernel: [fix](https://gitlab.eng.vmware.com/ayelam/shenango/-/commit/bf5f2450a9c0b5f6957cafa761c46a818ede177c)
* Getting the memcached chart on the new kernel

# Week 6/12

## 07/30

* Spent all day on debugging the SEGFAULT while running Shenango with a newer DPDK 20.11
* Some shenango notes
  + Sets up shared mem for exchanging packet buffers between iokernel and the runtime, which it registers with NIC as well avoiding packet copying.
  + For registering this piece of shared memory with the NIC (instead of leaving buffer allocation to PMD), it had to edit mellanox pmd code to expose memory registration functionality to the dpdk application layer - hence the dpdk patches.

## 07/29

* **1:1 Meeting notes**
  + Some interesting discussion around which parts of process memory should go in the remote memory. Having the whole process memory remote-able would be nice because of the flexibility it offers from the disaggregated OS perspective but it might be too expensive to put segments like stack in remote memory. TODO Talk to Marcos?
    - Or is it really? Assuming there are a few stacks (one per each kernel thread) and stack accesses are sequential (either grows or shrinks as the program goes), perhaps there won’t be too many cache misses (not as much as heap?).
    - But this logic may not work in case of user-level threading where there might be 100s of threads/stacks.
* Racklette notes
  + Marcos raised an interesting question about how two types of network traffic can be concurrently supported: the memcached traffic over shenango/tcp and the Kona memory traffic over RDMA. This must be somehow supported as memcached was run on Kona before and would have faced a similar problem.
  + Related suggestions for scheduler activations: Hyperupcalls and asynchronized page faulting. Also talk to Nadav Amit.

## 07/28

* Shenango integration plan with Kona
  + Memcached uses malloc() for its kv storage thru its slabbed allocator. In my experiment, memcached hits 2GB of memory, but that depends on the client, key distribution, etc. We can just interpose this malloc with Kona where only the heap goes on to the remote memory.
  + Kailua paper says that userfault are triggered for all anonymous page allocations. That’s how shenango allocates its working memory as well, would that create conflict?
* Shenango on latest kernel
  + I’ve isolated the seg fault to a particular piece of code but still don’t know why it is happening. It certainly has to do with upgrading the dpdk version because the same issue happens on the earlier 4.15 kernel with upgraded dpdk. I need to read dpdk [release notes](https://doc.dpdk.org/guides/rel_notes/release_20_11.html) between 19.11 and 20.11 for any breaking changes.
* TODO: What about the direction of memory affinity for threads? Where would that fit in this story?
* TODO: Kona vs Fastswap
* Racklette slides

## 07/27

* A brief pass over related scheduler work
  + *Callisto: Co-Scheduling Parallel Runtime Systems*

A userspace runtime that provides upcalls similar to scheduler activations and required constructs to build user-level schedulers. It can run frameworks such as OpenMP that our recommender app uses. I could not find the code.

* + *Cilk: An Efficient Multithreaded Runtime System*
  + *Shinjuku: Preemptive Scheduling for μ-second-scale Tail Latency*

Introduces lightweight hardware-interrupt mechanism to preempt a user-level thread (without trapping to kernel, or *VM exit*) in matter of microseconds. With this new capability, threads do not have to run-to-completion or cooperatively yield back to the scheduler, allowing user space scheduler to cut off long running threads and help with tail latencies for high dispersion workloads.

* + *Grappa: A Latency-Tolerant Runtime for Large-Scale Irregular Applications*
  + *ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks*

Need to take a deeper look but I think the key idea is to have per-core work queues/threads and efficient work stealing to eliminate HOL blocking and preserve work conservation.

* + *Other runtimes: libfiber, Intel’s Cilk++, Intel’s C++ Threading Building Blocks (TBB), OpenMP*
* A case for scheduler activations for building efficient schedulers on remote memory
  + **Assumption**: Page faults are a lot more common with disaggregated memory, especially as the size of local memory cache is reduced.
  + One way to hide remote memory misses is to stick to userspace for remote memory access, like AIFM. This way, the misses are handled in Userspace which the scheduler knows about, and can switch out the thread and run a different thread. However, this loses transparency and is not backwards-compatible.
  + To keep native virtual address space for remote memory, misses will incur page faults which take away kernel threads from application schedulers which affects throughput. In fact, I hypothesize that if I take memcached charts from remote swapping systems like Fastswap, I’m guessing that a lot of throughput degradation is not due to the stalling caused by remote memory miss but rather taking away the kernel thread from the application during the miss. (How can I verify this hypothesis?)
  + Can the upcalls also serve as carriers of page fault information and hence provide the scheduler with the page information that is normally hidden from the scheduler? (notwithstanding Kailua work)

## 07/26

* Upgrading Shenango to Kernel 5.9
  + Step 1: Upgrading DPDK to a version that works on the latest kernel i.e., 20.11. Working on updating the [patches](https://gitlab.eng.vmware.com/ayelam/shenango/-/blob/master/dpdk.sh). **TODO**: Also submit this patch to the original shenango repository.
  + IOkernel with dpdk runs with the new kernel!
  + But the runtime sees a segmentation fault. Debugging…
* Walking through the Shenango memory code to understand the error and also to get an idea of whether/how it can be integrated with Kona.
  + According to the paper, apps need to talk to the kernel to allocate memory so memcached is probably using malloc for kv memory. (Confirm)
  + TODO

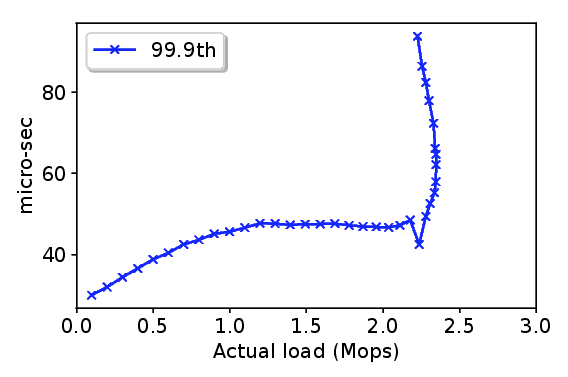
# Week 5/12

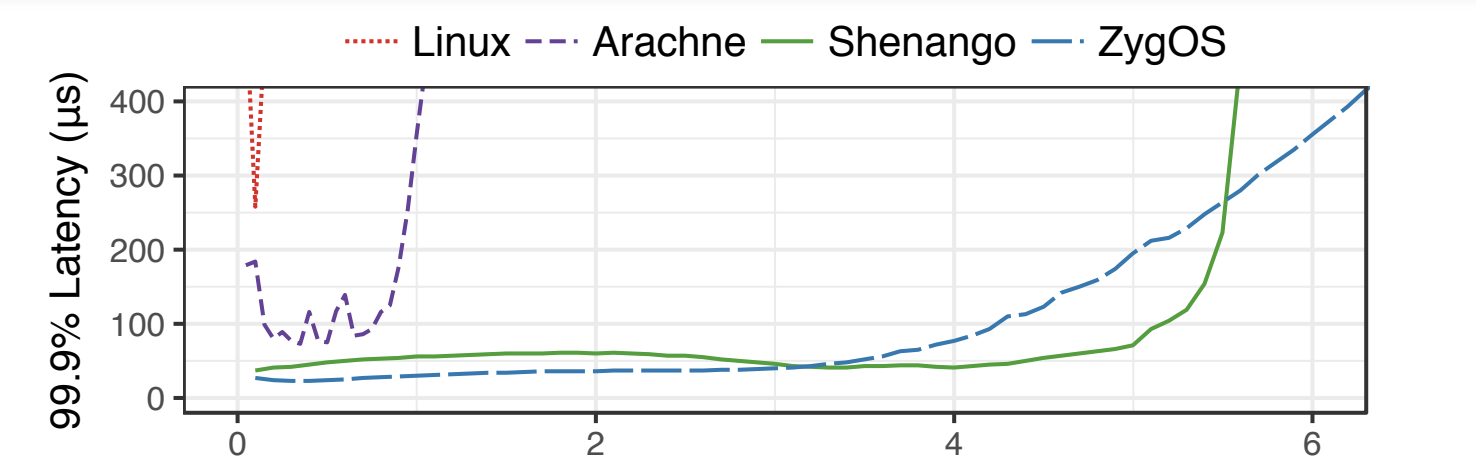
## 07/23

* Scheduler Activations Notes
  + Schedulers like Shenango and Arachne sidestep the challenges of blocking threads. While Shenango implements its own net stack to avoid socket calls for networking (Arachne does nothing to avoid even this), both are susceptible to implicit blocking events like page faults. The solution to this, and all other kinds of blocking kernel calls, was provided by scheduler activations where the framework gives the application a new processor when it loses one due to blocking.
  + This requires the Kernel - when an application blocks and traps into it - to store the *user-thread* context and give it back to the user-level scheduler along with the processor (kernel thread) that it has lost to blocking. The user level scheduler can then schedule other user threads on the processor, without a net loss of processor resources. The mechanism by which the kernel does this is called a “scheduler activation” which is like a vessel that provides/revokes kernel threads to/from userspace scheduler (userspace provides entry point and handlers for different kinds of activations).
  + Are there any real-world systems that implement scheduler activations or the like? It seems to have been implemented in some OSes but has been abandoned in favor of the usual 1:1 kernel threading - why? Are there any alternative solutions to the blocking issue? According to Geoff, it's just that the performance benefits didn’t prove to be worth the additional complexity.
  + Windows’ [User-Mode Scheduling - Win32 apps](https://docs.microsoft.com/en-us/windows/win32/procthread/user-mode-scheduling) feature is based on scheduler activations. Could not find a similar feature on Linux though :-(

## 07/22

* First look at Memcached’s throughput numbers with our setup. I kept the parameters of the run (e.g., client request distribution, number of server and client cores, client threads, etc) same as Fig 3 of Shenango. The main difference is the number of client servers, of which we have only one while Shenango employed four or five; I believe that’s where the throughput difference is coming from.





* The latency numbers look very similar (~50 µs in both cases) until of course they hit the max load after which they rise up to the heavens.
* My goal is not to match the numbers in the paper, I think I have a good enough baseline to work with. I’ll move on to the Kona integration now.

## 07/21

* **Integration with Kona: Expected Challenges**
  + (**Conceptual**) When a thread incurs a page fault, the underlying kernel thread is blocked rendering the core inactive for other user threads in implementations like Shenango. Current user schedulers ignore such blocking due to low page faults which is usually the case with general servers (otherwise the performance would tank anyway!) but page faults are more common in case of disaggregated memory so cannot be ignored. I feel like solving this is going to be the key contribution of our future paper.
  + (**Technical**) Would Shenango’s memory be amenable to remote memory integration. Last I checked, it is using memory based on pinned huge pages for at least a portion of the runtime.
    - TODO Deep dive into Shenango’s memory management
    - How does Kona handle mmapped anonymous pages that are possibly pinned?
  + (**Technical**) Kernel versions. Shenango works on 4.15 and my efforts so far to run on the later kernel have failed. But we need 5.7 at least to work with Kona.
* **Thought** The user schedulers reflect the principle of leaving policy to the userspace and mechanism to the kernel w.r.t. efficient core allocation. Kailua does a similar thing for remote memory management. Is it natural to combine these two? Think more about this. TODO
* TODO Look at some of the scheduler papers you gathered from Arachne that use the idea of asynchronous system calls to avoid thread blocking. Would a similar solution help with page fault related blocking?

## 07/20

* Debugging client-side “synthetic” application which is written in Rust, for crying out loud! The app also runs on Shenango, so I set up iokernel/runtime on b1607. Presumably, it uses Shenango’s netstack to connect to the memcached server. Right now the application boots up and seems to be “working” but forever without results.
  + The issue was that the mellanox nic ports on b1640 are wired in a different way compared to other servers, and shenango by default binds to the first dpdk port that is available. Edited code so that shenango takes any dpdk port that I specified and after binding to proper dpdk port on each server, client and memcached server seems to be exchanging messages!
  + Now the client prints some latency numbers WAHOOO!!
* But the network traffic monitored using ifconfig does not change at all during the experiment. Is it an ifconfig problem or ours?
* Shenango’s Net Stack
  + Shenango iokernel core binds to a NIC port and sends/receives ethernet packets which it exchanges with the runtime through shared memory. The runtime implements the whole net stack from ethernet to transport layer, including ARP cache etc. The whole setup bypasses the traditional net stack and so interestingly does not use any usual dns names, ip and mac addresses of servers, etc.

## 07/19

* Memcached server up and running. Setting up clients now; its requiring some fixing on the other machines. Using b1607 as client, debugging on the client side; almost there!
* Think about how Shenango’s memory management could affect our planned integration with Kona
  + Thread stacks are maintained in slab memory which is allocated from huge pages using mmap. What else is maintained in slab memory?

# Week 4/12

## 07/16

* Shenango code pointers
  + Takes extra care to strictly separate memory allocation by NUMA nodes
  + Always wondered how user-space threads are context-switched (saving the registers, maintaining stack and instruction pointers, jumping to a different thread, etc). Shenango does it in an assembly file which is surprisingly not that complicated: <https://gitlab.eng.vmware.com/ayelam/shenango/-/blob/master/runtime/switch.S>. It’d be nice to try this out for myself sometime.
  + Shenango’s runtime implements the entire net stack, and this includes allocating tx and rx buffers from *slabbed memory that it allocated from huge pages using mmap during initialization*. How would such memory affect integration with Kona?
  + I like the (generic?) tcache abstraction that is used to maintain a pool of resources like threads, stacks, connections, etc. to be reused without the overhead of creation. <https://gitlab.eng.vmware.com/ayelam/shenango/-/blob/master/base/tcache.c>
* Finish **Arachne** and add **notes**
  + Arachne is a user-level threading that provides much fine-grained parallelism compared to Kernel threads that are a bit rigid. That’s not new of course, but I think the key contributions are 1) a more efficient threading primitives (e.g., thread creation) that takes cache misses into account and load balancing threads across cores and 2) a core arbiter to automatically allocate/deallocate cores to application by monitoring utilization, load factor, etc. It also provides a nice detail on implementing threading primitives.
  + The difference between Shenango is that Shenango takes it a step further in estimating demand by looking at the packet queues that more accurately reflect computational demand of the application (like request traffic in memcached). However, both follow a co-operative multithreading model where preempting is not the common case and thread must yield is not expected to block - kinda hard to do that with page faults, where our opportunity is. *Is Arachne a better/easier system for us to work with?*
  + Couple of interesting related work to read on threading:
    - Cilk: An efficient multithreaded runtime system
    - Carbon: Architectural support for fine-grained parallelism on chip multiprocessors.
    - Capriccio: scalable threads for internet services.

## 07/15

* Setting up memcached on Shenango; looking into its code and how they ported it to Shenango.
* Interesting OSDI/ATC’21 papers
  + [The nanoPU: A Nanosecond Network Stack for Datacenters](https://www.usenix.org/conference/osdi21/presentation/ibanez)
  + [Naos: Serialization-free RDMA networking in Java](https://www.usenix.org/conference/atc21/presentation/taranov)
  + [One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory](https://www.usenix.org/conference/atc21/presentation/zuo)
* Couple of really old gang/co-scheduling papers in HPC that looked interesting
  + <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=845971>
  + <http://web.stanford.edu/~ouster/cgi-bin/papers/coscheduling.pdf>

## 07/12, 13, 14

* Mostly Shenango troubles. First kernel version issues, and later getting it to run with CX-5s. Created gitlab repos for my workspace: <https://gitlab.eng.vmware.com/ayelam/rmem-scheduler>
* Shenango is properly running! Working on memcached now.

# Week 3/12

## 07/07-08

* Debugging Shenango on my setup. Running into some NUMA membind errors.
* **Racklette weekly notes**
  + Floated the Shenango + Kona idea to start the work of co-scheduling green threads on remote memory. We are starting with memcached because Shenango already supports it but would Shenango’s model work for the recommender app in a similar way? [Gerd] Shenango seems custom-tuned for memcached-like applications and may not generalize?
  + I’ll start with one compute node running the memcached server and one remote memory node. But we may want at least two memory nodes if we want to schedule based on memory contention in either node (because you can schedule threads so that the contention on either server is uniform). In general though, we need a way to map threads to the memory they are using i.e., have some sort of memory affinity information.
  + [Marcos] In a simple case, we can associate each piece of memory to the thread that alloc’d it. We can either change the app and have the threads malloc() the memory they work with or use the first touch policy in Linux to be transparent. But allocation affinity may only give us a limited view of the actual access affinity. Perhaps we can use page faults as another proxy which is more correlated with the actual accesses?
    - TODO Learn more about first touch policy and memory affinity problem in general
  + [Gerd] We can look at libNUMA APIs for inspiration on memory affinity, and also some gang scheduling papers from HPC since the whole idea seems related.
    - TODO Collect some papers
* **MIND paper: In-network mem management for remote memory**
  + Uses a P4 switch to implement address translation and MSI cache coherence protocol.
  + The memory management aspect is similar to LegoOS in that the compute node cache is virtually indexed and address translation is done at the memory nodes. The routing part is done on the switch which only needs to know the mapping between virtual memory ranges (which are kept large to minimize metadata) and the memory nodes.
  + The key contribution is providing shared memory for compute scaling, and implementing MSI cache directory protocol on the switch to that end. However, the results did not convince me of the idea of scaling out cache coherence. The switch may help cut down the RTT by a factor but the latencies for maintaining the directory are still OOM more than local cache coherence traffic and MSI is too strict a consistency model to make it performance-feasible for most applications. Naturally, the applications that are evaluated are the ones that show less memory sharing like TensorFlow.
  + I do like the idea of bounded splitting to overcome the switch space limitations in storing the cache directory but at the same time allowing a smaller unit of sharing to avoid false invalidations. I wonder if it would really scale for more interesting applications. This one was submitted to SOSP 21 but I’d be surprised if it gets in, let’s see!

## 07/06

* List of people you should talk to: [Remote memory](https://research.vmware.com/projects/remote-memory) + [Gazelle](https://research.vmware.com/projects/gazelle-towards-predictable-low-latency-networks)
* Preparing slides for racklette: Project direction + PL2; no time for anything else today.

## 07/05

* Run memcached on Shenango + Kona
  + How to integrate a new app with Kona? Is preloading alloclib enough?
  + It’s better to get all related repos at once and build them using instructions at: [shenango/all](https://github.com/shenango/all). It provides a quick way to replicate the paper’s experiments. But it did not seem to be built for the current kernel version and I’m seeing some errors. I went back to the 18.04 kernel but the errors persist.
  + Modifying [experiment.py](https://github.com/shenango/all/blob/master/experiment.py) to run just memcached on Shenango. It fails to start up due to a huge page issue.
  + I may need to understand memcached better. What do the terms barrier-leader and barrier-peer mean in this context?
* **1:1 Meeting Notes**
  + When thinking of scheduling on the compute node, think of not just signals within the server but from the rack e.g., from the network fabric and the contention on the memory servers. How do we associate the thread with the page it faulted over? AIFM scheduler knows which object the thread is requesting based on dereferencing smart pointers. With Kailua, we can find (maybe there already is) a way to associate the user thread with the page fault, and since page fault handling is in user-space, the scheduler can be in touch with the page fault handler and make informed decisions.
  + **Question**: Well, how does this information help the scheduler? One example we discussed was how predictability of when the page request will be serviced can be used by the scheduler to schedule threads in some deterministic way to maximize utilization/throughput.
  + We can also think about lock contention when memory is locked from multiple compute nodes. For example, instead of a thread polling on the lock that is maintained at the rack controller, an integrated scheduler can be informed by the rack controller when the lock is released which then schedules the thread which waits in the queue until then, thereby saving some CPU.
  + Just like balancing between application and eviction threads in AIFM, I think similar idea can be extended to resource usage of these two “classes” of work (i.e., main app work vs housekeeping) to various resources. e.g., the network which carries both application request traffic and remote page traffic.

# Week 2/12

## 07/02

* Try Shenango + Memcached + Kona
  + Looking at Shenango code: [shenango/shenango](https://github.com/shenango/shenango)
  + Memcached port for Shenango: [shenango/memcached](https://github.com/shenango/memcached)
  + Has any remote memory papers used memcached before? **Yes**, seems super common actually. **Examples**: Infiniswap, Fastswap and Leap. Refer to these works for precise configurations if needed later.
* Why Shenango, and not Caladan?
  + Shenango is a simpler framework with the green thread support we want.
* AIFM used Shenango and had an opportunity to perform coscheduling. Why didn’t it?
  + It does, actually. Section 5.4 does coscheduling for essentially balancing between the application threads vs. remote memory evacuation threads. I have never appreciated this before but it cleverly uses dereference scopes as a way in which application threads can let the scheduler know that they are holding to a certain piece of remote data and prioritize threads whose data is currently local. We will face this question with our scheduler as well, but maybe there is a better way? **TODO**: This needs more analysis!
* It is interesting to note that the network now supports both request traffic and remote memory traffic, both of which affect (end-to-end) throughput/latency of memcached. Co-scheduling these two traffic classes may be key to improving throughput - looking at it from a completely network standpoint.

## 07/01

* **1:1 Meeting Notes**
  + Talk to Jiacheng for a quick ramp-up on running the DLRM app. (**On Hold**)
  + We agreed that starting with Shenango/Caladan as our scheduler is better than starting from scratch. On that note, I am going to try and run Shenango + memcached first on a regular remote memory framework i.e., Kona without user-fault support.
* **Racklette Meeting Notes**
  + Jiacheng presented the memory management API. See [here](#_aw39ssasij2s).
  + **Thought**: After you figure out that Shenango doesn’t work well for remote memory, (how) can you make the Shenango scheduler far memory-aware? How can a thread be linked to the memory it is using in a way that the runtime is aware of it? In other words, if a thread faults on or is about to page fault/cache miss on a particular piece of remote data, would it help for the scheduler to know about which page it faulted on?
  + Learn a bit about CXL - what kind of interconnect is it?
* More notes on **Caladan and Shenango**
  + More related scheduler **papers to read**: Arachne, Snap, Shinjuku, PerfIso, Heracles.
  + Caladan resorts to core allocation to *manage* interference/contention at various resources like LLC and memory bandwidth. The LLC case is interesting; since the cache is not in runtime’s control, it resorts to just changing core allocation to control the cache space. In the case of remote memory, we could imagine a similar interference stemming from the local DRAM cache, which in this case can be explicitly managed by the runtime with better control over the cache allocation?

## 06/30

* Wrapping up **Caladan and Shenango**. Add notes when you’re done.
  + The need for μs-core allocations in these papers seems to stem from the need to react to *external* load variations without incurring high tail latencies. We should have a similar application if we are following their example. The papers generally use memcached and similar server applications. Would the Recommender inference rise up to the task of latency-sensitive benchmark?
  + Both these papers required fine-grained control signals to estimate load without asking the application, hence they provide custom runtime with user-level lightweight threads that applications can use to chunk (represent) their work (RThreads?). These threads also let the frameworks move the work around more efficiently. We will probably need to present such custom runtime as well. Kona at the moment doesn’t have such support: is the first step adding one?
  + These systems offer slightly different views of the source of tail latencies. Shenango treats tail latencies as stemming from slow reaction of the runtime to the load, and hence making the allocations more fluid. Caladan takes it further by presenting it as a consequence of the interference from other apps at different resources, and hence focuses on identifying interference and offering performance isolation. Both these can be valid problems in a remote memory setting as well, although we probably want to focus on improving scheduling like Shenango rather than performance isolation from other applications.
  + **A scoped problem**: (Caladan for remote memory, *not exactly Caladan for Rack*) Take Caladan and run it on a remote memory system like Kona, look at the tail latencies. Clearly, Caladan’s scheduler would need to account for page faults/fetching while scheduling and we can think about/experiment on how to do that from there. But this only works on server-level, not rack-level, and does not involve Kona (at this point?). Speaking of, can we possibly integrate Caladan and Kona?
    - **Rebuttal**: Is it too soon to worry about tail latency as a metric? i.e., remote memory applications still struggle with throughput which generally comes first. In other words, should we think about optimizing remote memory for data-crunching apps before latency-sensitive ones? (I feel this but can’t exactly point out why).
* Working with DLRM app on Kona
  + DLRM original paper: <https://arxiv.org/pdf/1906.00091.pdf>
  + Looked at *mbench* code; pretty simple benchmark measuring *malloc/free* and *rmalloc/rmmap/rfree* calls. Browsed some Kona core files like *klib.c* that implements the rAPI.

## 06/29

* **1-1 Meeting Notes**
  + Jiacheng is working on scaling Kona to multiple *compute* nodes. This presumably means providing an application with the same virtual (remote) address space on all the compute nodes so that it can run anywhere. However, consistent sharing is not supported so the app is responsible for maintaining consistency for which it provides basic synchronization operations supported by the rack controller (the model seems to be inspired by the *coarse-grained* synchronization in Marcos’ Remote Regions paper). Even if the app is not distributed, we can now *transparently* preempt the app to another *server* if needed. This might allow some interesting far memory-aware cluster-level scheduling. He also provides a compute node API that lets apps register with multiple nodes and know which node the current instance is running on so it can act accordingly when accessing shared remote memory. Exactly what granular is a “node” though: is it a thread, process or a server? A smaller unit e.g., thread might allow better scheduling within the server as well but it might not let the app exploit the local cache coherency on a single server.
  + Is Recommender the right application for us? Previously-used benchmarks: Spark, Memcached, Redis, TensorFlow, Intel LinPack, Pandas DataFrame, etc.
* Emailed Mosharaf for Justitia paper
* Looking at Calidan and Shenango

## 

## 06/28

* Goal today: Run dlrm app with Kona ❌
* Understood user faults concept thanks to Kailua paper. Browse code!
  + Do I need to understand Kona code in detail? Is that relevant to my project?
* Organize thoughts on Racklette. What directions can we explore?
* TODO Find a couple of people to talk to this week. Sujata & Marcos?
  + Get help from Radhika
* TODO Read a rack-scale paper later today: ~~R2C2~~ Racksched ✅
* Went through the [Racklette scheduler meeting notes](https://docs.google.com/document/d/13Ig6eoHNZVFZsL-UpgmmgQUn2Zk5dbaaJcoeYgtDQVs/edit?ts=60d0fae9#); makes more sense after the racklette paper. So many open questions and assumptions yet to be made! Some meta-ones:
  + How long has this idea been cooking at VMware? Who is leading the effort?
  + What’s the retreat thing? The discussion looks impressive!

# Week 1/12

## 06/25

* Upgraded OFED on b1640. RDMA ping works!
  + Turns out previous error 12 was due to using the wrong interface, or, the interfaces and ip addresses on b1634 seems to be wrongly swapped.
* Kona runs! Yay!
  + Both basic and mbench apps.
* Onto the recommender (dlrm) app! (I’ll start this on Monday)
* Write down Racklette and Kailua initial thoughts/notes
  + Notes in Remarkable!

## 06/24

* Fought with RDMA ping test. Turns out Gerd was using/tweaking ofed stack on b1634. Trying to get another machine.
* Reading Kailua. Seems to be exploring the transparent-yet-also-application-integrated dimension to enable adoption but keep the performance benefits of app integration. (nice to see something answering the question that I asked in the future work of my quals report)
* Working with b1640. Need to upgrade OFED.

## 06/23

* Managed to upgrade Ubuntu!

ILO access: <https://sc2-hs2-b1630-ilo.eng.vmware.com/index.html#CIMC> (only Firefox!)

* Servers with BF2 cards from Ming

w1-hs2-e2412

w1-hs2-e2413

* Kona build succeeded!
* But memserver and rcntrl failed to bind rdma socket
  + RDMA debugging: <https://community.mellanox.com/s/article/RoCE-Debug-Flow-for-Linux>

RoCE Ok. All the **Ib\_\*** tools fail with IBV\_WC\_RETRY\_EXC\_ERR (12) error

* Racklette notes
  + TODO: notes from remarkable
* Start Kailua paper

## 06/22

* Had a Kona tutorial with Jiacheng: [https://VMware.zoom.us/rec/share/iImEplhtLzd5THaS8OCmK9gyCHI1TeF4uoO8r\_6nlKHHYRHr\_Avd1y3Ta76Fmp0G.tDsJma-hp3i4nwh5](https://vmware.zoom.us/rec/share/iImEplhtLzd5THaS8OCmK9gyCHI1TeF4uoO8r_6nlKHHYRHr_Avd1y3Ta76Fmp0G.tDsJma-hp3i4nwh5) Passcode: x19Pti!J
* Environment set up! (SSH, VS code + Kona code ready) - trying to build.
* Working on upgrading the kernel version today…
  + Lots of bad turns...
* Finish reading the Racklette document and start kailua

## 06/21

* Intros and some resources

[Copy of Racklette](https://docs.google.com/document/d/1qCa_o6YSYwSWnfKynodvG9vYIPQr8zibaycxLU_ZbaE/edit?ts=60d0fb73#heading=h.1mq6458bolhl)

[Racklette scheduler meeting notes](https://docs.google.com/document/d/13Ig6eoHNZVFZsL-UpgmmgQUn2Zk5dbaaJcoeYgtDQVs/edit?ts=60d0fae9#)

[Gazelle Testbed](https://docs.google.com/document/d/1m2suV-mhhDgPkhk2h32hJoiY4heNFYXnjXEjS0bgFaA/edit#)

**Kona code**: https://gitlab.eng.vmware.com/icalciu/userspace-remote-memory

* Started reading Racklette.