# User Space TCP - Getting LKL Ready for the Prime Time

**H.K. Jerry Chu, Yuan Liu**
Google Inc.
1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA
hkchu@google.com, liuyuan@google.com

## Abstract

Running the networking stack in the user space is not new. The conventional wisdom is that the network stack must bypass the kernel in order to meet the performance requirements of a class of applications that demand super-low latency.

This paper describes an experiment we've undertaken to provide a production- strength user space TCP stack for a different use case inside Googles internal production network.

We choose a Linux based open source project called Linux Kernel Library (LKL) as a base for our effort, and have made significant contribution to it since late last year, improving both its quality and performance. During the time, we discovered a number of architectural constraints inherited in the LKL's current design and implementation, and gained valuable insights into the pros and cons of the different approaches to user space TCP.

## Keywords

user space TCP, Linux kernel library

## Introduction

It is a common belief that OS bypass is a necessity in order for the networking stack to meet the requirement of a class of applications from HPC to Wall Street that demand µsec-scale tail latency. Most of the commercial user space networking stacks are geared toward those applications. Moreover, special interconnects such as Infiniband and RoCE have been deployed inside datacenters replacing TCP protocol as the transport that boast lower latency, higher throughput while using less CPU cycles.

We are faced with different challenges. First there is a set of legacy use cases where TCP is required for backward compatibility. Second, Google's vast internal networks are designed to only carry internally generated traffic for a number of reasons. They range from tight security requirements, DOS prevention, to better resource control. Network traffic initiated externally and destined to services provided by Google must first be terminated at Google Front-ends (GFEs), which have been fortified to withstand adversary conditions, such as the ever increasing DOS attacks. But this comes at a steep computational cost.

With the rise of Cloud Computing, the boundary between internal and external networks have blurred. Nowadays packets destined for Google services may be initiated from a foreign stack installed by a cloud customer running directly inside Google's data center. If these "guest" packets created by untrusted stacks are allowed into our internal networks unchanged, and terminated directly by the Linux kernel TCP stack running on our internal servers, it poses a very high security risk. On the other hand, forcing all the guest packets to route through GFEs in order to subject them to the rigorous checks and filtering is undesirable, both from the cost and performance stand points.

Running a TCP stack in the user space to terminate guest connections provides a solution that much reduces our exposure to the security risk, from the whole OS kernel down to a single user process. It also provides us high velocity in deploying to the fleet any bug fixes we may find in the stack. Moreover, it allows more accurate and tighter resource control, and finer grained accounting, as compared to the kernel stack. For this reason different stacks employed by Google Cloud Platform (GCP) all provide their own user space TCP stacks, written in a variety of languages from C, C++ to Go.

As our cloud business grows, many of these small home grown TCP stacks, initially written for their individual needs, start showing signs of cracking due to their immaturity, and suffer quality, performance, or interoperability problems. This is not a surprise given the complexity and versatility of the TCP protocol. It may be possible to demonstrate an over-the-weekend TCP implementation that can perform some simple handshake when no packet is dropped by the network. It's yet another story to come up with one that can reliably inter-operate with the rest of the Internet.

This paper presents our effort in the past year to provide a production- strength user space TCP stack based on LKL. The rest of the paper is organized as follows: Section "Landscape of User Space Networking Stacks" gives a short survey of existing user space TCP stacks and our effort for finding one that best suits our needs. Section "Linux Kernel Library" introduces LKL and the various network configurations we use. Section "Minimizing Latency - the Holy Grail" details our efforts to cut down the latency of the LKL stack. Section "Large Segment Offload, the Key to Max Throughput" describes how we boost LKL's throughput performance. Section "Taming the Copy Overhead" describes our effort to cut down the number of copy operations. Section "Debugging Facility" describes a very useful facility we've added to make it easy to retrieve MIB counters from the LKL stack. Section "Linker and Loader Issues" documents an example of challenges in compiling kernel code as a user library. We conclude with a TODO list for the future.

## Landscape of User Space Networking Stacks

As described previously, given the complexity of the TCP implementation and our requirement for a high quality, production-strength TCP stack, we decided against writing one from scratch, and started looking in the existing open source space.

There are quite a few user space networking stacks out there. We have a preference for a Linux based one, both for its quality, performance, rich feature set, where Google has made significant contribution, and our familiarity with it, since Google's data centers mostly run on the Linux OS.

Library operating system (LibOS) [6] first caught our attention. It seemed to be a good candidate, based on Linux, focusing only on the networking stack hence likely requiring a smaller footprint, and possibly avoiding the complexity and performance overhead of a full-blown OS. It confines all the changes to an architecture directory `arch/lib` hence allowing it to adapt to future kernel releases.

User-mode Linux (UML) [2] was brought up as another candidate. But it does not seem to match our model of linking the networking code with the application. It may be possible or even not difficult to do so but we do not have enough expertise to assess further. Some other team inside Google had previously evaluated UML and decided the ptrace syscall overhead was too high before they went on to create their own stack. (Unfortunately, as we discovered later, syscalls in LKL are quite expensive too.)

After an initial evaluation of LibOS, we discovered the project seemed still in an early stage hence implementations in many areas seemed incomplete resulting in many bugs for our use case. This has been confirmed by the author of LibOS that it was targeted initially for a network simulator to run Linux TCP code, hence the various race conditions we encountered when using it in a different environment were not an issue for the simulator. Also it confirmed our worry that it can be more difficult and error-prone to try to carve out a piece of code as complicated as the network stack, which is fully integrated with the rest of kernel, and use it as a stand alone piece of software, rather than taking the kernel as a whole. A lots of kernel internal interfaces will have to be exposed and replaced by new code in the user runtime, which can compromise the original high quality of the monolithic kernel.

On the other hand, LKL seemed to be much further along in development. Like LibOS, it was also done as an architecture port to the Linux kernel hence changes are isolated under an architecture directory `arch/lkl`. If it ever makes into the upstream kernel repository, the maintenance cost will be reduced to a minimum.

But as we discovered later, the higher quality resulted from taking the whole kernel in one piece also comes with a steep performance price.

## Linux Kernel Library

The open source project was created by Octavian Purdila[1] with a detailed paper published in 2010 [5].

Like UML, LKL was done as an architectural port of Linux. It relies on a set of "native operations" supplied by the host OS to glue the LKL kernel to the host environment. Its architecture is illustrated in Figure 1.

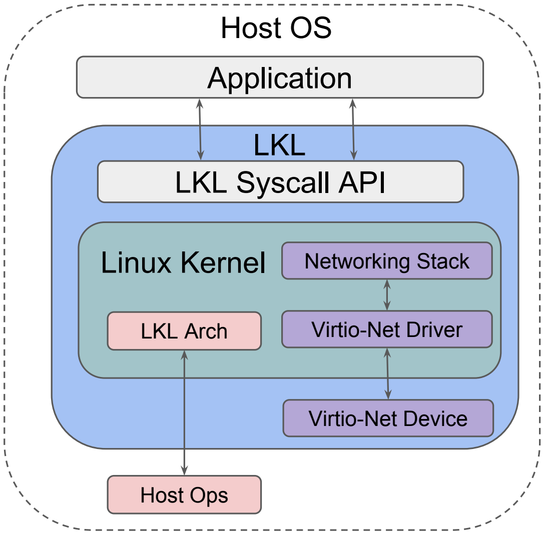

Figure 1: LKL architecture.

For the networking stack inside LKL to communicate with the outside world, one simply plugs a virtio-net virtual "device" into LKL. Currently there are quite a few virtio-net devices available for LKL already. The list includes tuntap, raw socket, VDE, dpdk,..., etc. We've written a RDMA based device to allow LKL to run directly on top of RoCE capable NICs, bypassing the host OS all-together. This is shown in Figure 2.

Two other network configurations pertaining to our work are illustrated below. Figure 3 illustrates our first application where LKL is linked with a TCP proxy to terminate connections from VM guests. The proxy then relays the guest requests to Google's backend servers through the usual socket or other networking API provided by the host OS.

Figure 4 shows the second configuration that simply runs LKL on top of a TAP device to inject or retrieve packets to/from the host kernel. This configuration allows two socket applications, one running on top of LKL, the other running on top of the host OS to communicate, and is easy to setup so we use it often for debugging and regression test purpose.

After flushing out many bugs initially, we managed to link LKL with our internal applications and it appeared to work pretty well. But we quickly discovered its performance overhead was much higher than the native stack. The following sections described some of our efforts to narrow the performance gap with the native OS.

[1] octavian.purdila@intel.com

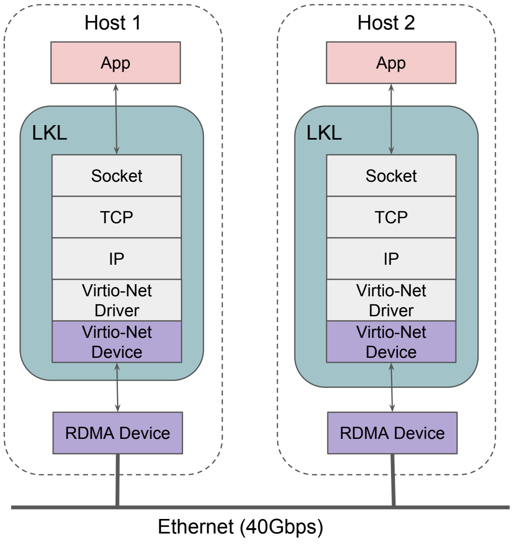

Figure 2: LKL on top of RDMA.

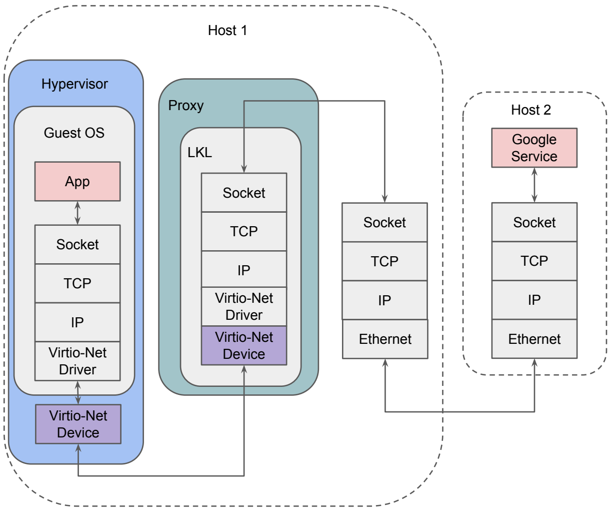

Figure 3: LKL TCP proxy.

## Minimizing Latency - the Holy Grail

Our initial performance test showed LKL exhibited 4X latency as compared to the native, host stack. This did not come as a complete surprise due to LKL's design as explained below.

Various parts of the kernel code have a general assumption that all the threads executing the kernel code are under the direct control of the kernel scheduler. Some of the kernel code even has dependency on threads running with a kernel stack, and will access the stack frame based on the structure of a kernel stack. Since LKL retains much of the kernel code including the kernel scheduler, threads created by the application and scheduled by the host scheduler can not safely execute the LKL kernel code directly. Instead, a corresponding "shadow" LKL kernel syscall thread must be created for each application thread to handle all syscalls on its behalf. Though there is no longer a mode switch from user space to kernel, a context switch from application thread to LKL kernel thread is required.

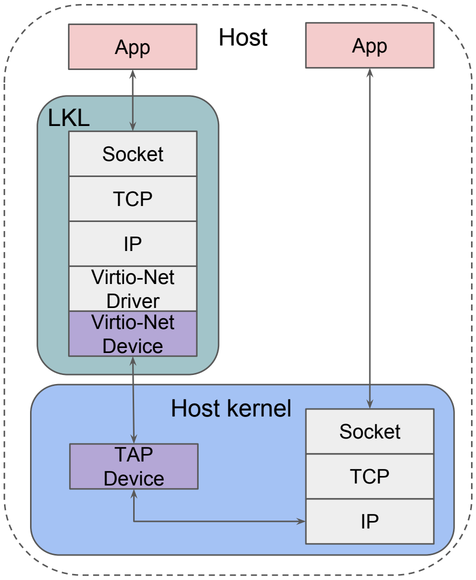

Figure 4: LKL on top of TAP.

Figure 5 describes the process of running a LKL syscall. Any host thread calling LKL syscalls must trigger an LKL IRQ to switch to LKL "kernel space". LKL IRQ is handled by LKL idle thread or any LKL kernel thread that re-enables IRQ. The IRQ handler wakes up the LKL kernel syscall thread corresponding to the original host thread making the syscall to actually run the kernel syscall code on its behalf. On completion, it unblocks the host thread through a host semaphore.

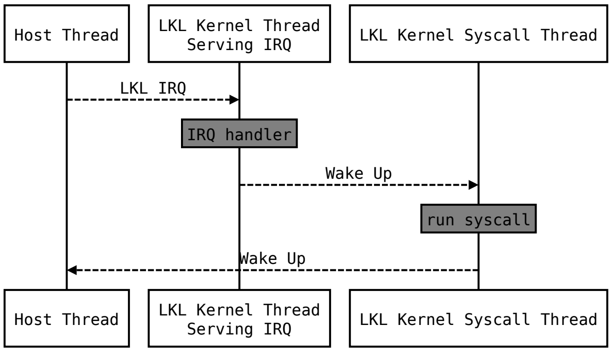
Host Thread
LKL Kernel Thread Serving IRQ
LKL Kernel Syscall Thread
LKL IRQ
IRQ handler
Wake Up
run syscall
Wake Up
Host Thread
LKL Kernel Thread Serving IRQ
LKL Kernel Syscall Thread

Figure 5: Process of running a LKL syscall. Each dashed line represents a context switch.

As shown above, each LKL syscall incurs three additional context switches; each costs several microseconds. It's much more expensive than the cost of native syscalls (e.g., non-blocking socket read typically takes less than 1 µs).

### Pin Threads to a Single CPU

One can often get a performance boost by pinning threads to CPUs to take advantage of better cache locality and to avoid the expensive inter-processor interrupt (IPI) calls. Our benchmark shows context switch using semaphore takes ~2.5 µs on average but only ~1.5 µs if on the same core. When we pin all threads onto a single CPU core using `taskset(1)`, we see big improvement on `TCP_RR` latency. However, `taskset` is not always the best option in reality. The application may contain parallel threads that best perform with multiple cores.

We decided to pin LKL kernel threads with `sched_setaffinity(2)` internally without involving the host threads. Thus we benefit from the faster context switch on the same core without losing any parallelism because LKL is anyway a uniprocessor architecture.

### Direct Syscall

LKL syscall is much slower than the native syscall due to the three additional context switches as described above. A simple benchmark shows LKL `getppid(2)` takes roughly 10 µs while host only takes 0.4 µs. Even with `taskset`, it takes 3 µs. Can we do better?

As mentioned before, some part of kernel code, especially in the scheduler area relies on the fact that all threads (or LWPs to be more correct) executing the kernel code are originally created by the kernel. Specifically every thread executing the kernel code must have a valid `task_struct`. Since every host thread making a LKL syscall for the first time will cause a shadow LKL kernel thread to be created, we'll let the host thread borrow its shadow thread's `task_struct` as follows:

- Lock LKL. A global mutex lock is added and any thread that runs LKL kernel code must acquire this lock.
- Save current `task_struct` (of whoever happens to be running in the LKL kernel) and IRQ status.
- Change `current` to point at the shadow `task_struct` and enable IRQ.
- Run syscall.
- Restore current `task_struct` and IRQ status.
- Unlock LKL.

One may notice we don't adjust the scheduler status in LKL so in the run queue, it's still the original tasks. That means the host thread can't block in the LKL kernel. To make blocking syscall work requires changing the generic scheduler code unfortunately, to hijack `__schedule()`. I.e., instead of letting the host thread execute the LKL kernel scheduler code directly, we make it block on a host semaphore, which will be posted by the shadow thread when LKL schedules it again.

To achieve the best latency number, dyntick-idle mode is disabled in LKL. Otherwise the LKL idle thread must be woken up after every direct syscall to pick up any possible changes to the system (e.g. another thread is waken up by the syscall). By disabling dyntick-idle mode, the wake up is done by LKL timer ticks, which can happen in parallel. This optimization saves several microseconds at the cost of waking up LKL every 10 ms (HZ=100).

With direct syscall, LKL `getppid()` now takes only 0.2 µs now.

### Latency Evaluation

Figure 6 shows the 1-byte `TCP_RR` latency of different approaches, using the kernel-bypass configuration as shown in Figure 2. With busy polling, which saves one context switch at each end, 33 µs is achieved compared to 23 µs of the host stack. That's within 1.4X of the host. Without busy polling, 40 µs (1.8X) is required. Some heuristic can be applied here to avoid busy poll indefinitely.

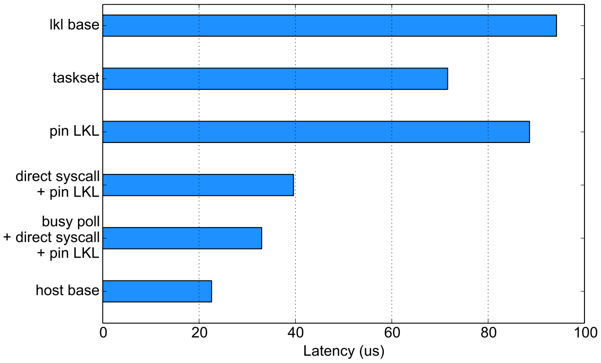

Figure 6: 1-byte `TCP_RR` latency. `lkl base`: baseline of LKL using RDMA; `taskset`: pin all threads to a single CPU; `pin LKL`: pin only LKL kernel threads; `direct syscall`: allow host thread directly to run LKL syscall; `busy poll`: busy poll incoming packet in RDMA virtio device; `host base`: baseline of the host TCP stack.

The biggest gap between LKL and the host is the IRQ handling of incoming packets. In LKL, IRQ is handled by the LKL idle thread, or any LKL kernel thread that re-enables IRQ, which means a context switch is needed, hence much slower than the real hardware IRQ. A direct IRQ handling mechanism is being experimented during the preparation of this paper.

## Large Segment Offload, the Key to Max Throughput

Although having the whole kernel code in LKL incurs significant syscall and latency overhead as described above, it does provide a major upside - many kernel features are readily available and just work. E.g., GSO is provided in the kernel netdev layer independent of the network driver. Virtio-net driver also supports GRO. Either gives a boost to the throughput performance while remaining completely transparent to the rest of the device code, configuration, or the environment LKL is running within as shown later.

For the optimal throughput performance, it's best to maintain large segments, i.e., to avoid segmentation as further into the network path as possible. To avoid segmentation for virtio-net, we added to the LKL device the support of all flavors of segmentation offload. The device code that manages the descriptor rings was also enhanced to support both big_packets and mergeable_rx_bufs modes, in addition to the regular MTU as described later. With these enhancements, we enable large segments to travel between the guest and a local LKL client without ever being segmented or checksummed, thus greatly improve the throughput performance.

Figure 7 compares the netperf throughputs among a number of different offload configurations based on the setup described in Figure 3.

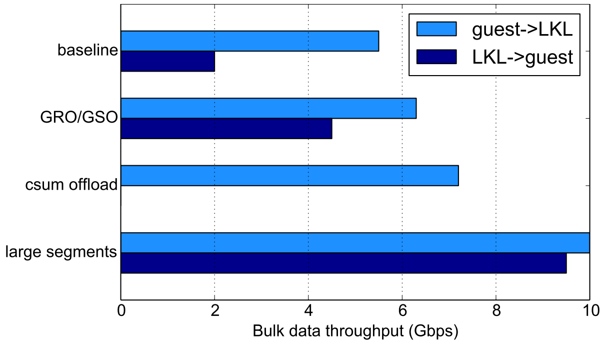

Figure 7: Single TCP_STREAM throughput with VM inside a 16-CPU container, LKL configured with 256MB memory. baseline: all offload features disabled; GRO/GSO: GRO/GSO enabled in LKL; csum offload: LKL GRO + csum offload; large segments: all offload features enabled; LKL uses the mergeable_rx_bufs mode

Linux kernel has over the years accumulated a large number of "offload" features in its netdev and driver layers. With the exception of GSO and GRO, most of the offload features require some form of support from the NIC devices. When we started to work on LKL a few months ago, its "virtual NIC" device, i.e., the "virtio-net" did not have much offload support. So we started with ~2Gbps throughput for the bulk data transfer for both directions between a Linux guest and the TCP proxy. We had to disable TSO in the guest because LKL's virtio-net device could not handle any segment large than the regular Ethernet MTU. The same applied to TX checksum offload.

After removing a few unnecessary copy operations in our code bridging the guest and LKL, we managed to triple the throughput from the guest to the proxy to 6.3Gbps. We discovered the GRO support in the virio-net driver also contributed - without it the throughput would be less at ~5.5Gbps. (But for some reason throughput went a bit higher at ~5.8Gbps if we disabled GSO in the guest.) Likewise for the LKL/TCP proxy to the guest direction, once we enabled CSUM offload (VIRTIO_NET_F_CSUM) in the LKL virtio-net device, GSO is automatically enabled by the LKL kernel and boosted the throughput from ~2Gbps to 4.5Gbps.

The above is one example of the advantage of linking in the whole kernel code - some of the desirable features like GSO and GRO just work with little or no change required.

Support of VIRTIO_NET_F_GUEST_CSUM device feature was added next so that the TCP checksum calculation can be bypassed on both sides between the guest and the proxy. This further improved throughput to 7.2Gbps. The next step was to allow large segments to flow end to end between the guest and the TCP proxy. For the LKL to the guest direction, support for another device feature VIRTIO_NET_F_HOST_TSO4 was added to the LKL's virtio-net device code. The guest side's virtio-net device in the hypervisor can already handle large receive segments hence was ready to go. This change more than doubled the throughput from 4.5Gbps to 9.5Gbps.

The other direction required more work. Virtio-net driver supports two different large receive modes. The big_packets mode is enabled by the device feature VIRTIO_NET_F_GUEST_TSO4 or like, which constructs descriptors in the RX descriptor ring as a set of chains. Each chain contains enough buffer space to accommodate the largest possible segment size, i.e., 64KB. The other, possibly more space efficient mode mergeable_rx_bufs, enabled by the device feature VIRTIO_NET_F_MRG_RXBUF, allows the device to consume as many descriptors as it needs for each inbound packet.

We added support for both modes and boosted the throughput to more than 10Gbps from 7.2Gbps.

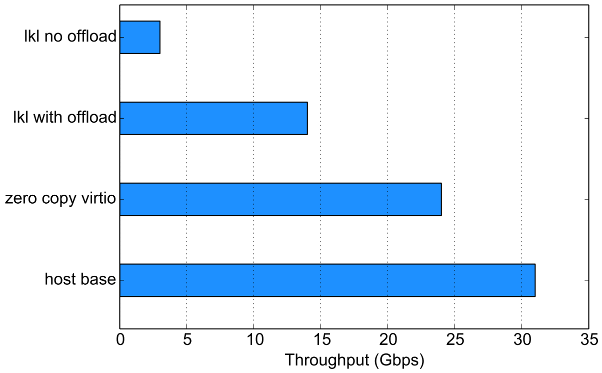

Figure 8: Single TCP_STREAM throughput of LKL on top of RDMA. lkl no offload: no offload; lkl with offload: csum offload + big_packets; zero copy virtio: avoiding copy in the virtio device; host base: baseline of host kernel with interrupt coalescing disabled.

To get a performance comparison between LKL and the host stack when doing bulk data transfers, we turned to the RDMA configuration since there is no equivalent host setup for the TCP proxy configuration. Here the effect of the offload support for LKL is even more pronounced, boosting throughput almost 5X from 3Gbps to 14Gbps over a pair of 40Gbps NICs as shown in figure 8.

We further improved throughput by posting buffers from the virtio-net driver directly to the RDMA NICs, avoiding the expensive copy operation. This gets the LKL throughput up to ~24Gbps, within 75% of the host throughput of 31Gbps. But at the max throughputs, LKL is close to max out one CPU core on both the TX and the RX sides, while the host only uses ~50% of one CPU core. This implies LKL burns more than 2.5X CPU cycles than the host to transmit/receive each Ethernet packet, still quite inefficiently unfortunately.

Note that this is only a rough estimate, as the device driver and NIC hardware features are likely quite different between what host IP stack configures, and the "RDMA" mode LKL uses.

## Taming the Copy Overhead

When moving data across different software components, memory copy is often the most convenient, and sometimes the only viable option. Using copying to transfer data between two distinct software components avoids any need for coordination between the two, sometimes residing in different protection domains. Once data is copied over, the buffer management code from the two software components can run independently to manage their own buffers, thus greatly simplifies their code logic.

The downside of copying is the significant cost of CPU cycles, especially when large segments are employed end-to-end. In the TCP proxy bulk-data throughput benchmark, there are four socket calls and copy operations involved. They are between the guest and the guest kernel, TCP proxy and the LKL kernel, TCP proxy and the host kernel, and finally the netperf or netserver and the host kernel. To transfer packets between LKL's virtio-net device and the one inside the hypervisor, two more copies are made. Hence a total of six data copies are invoked for each byte of data transferred; and the copy overhead ballooned to ~30% of the total CPU time after we enabled large segments end-to-end.

Removing the copy at the virtio-net TX direction is easy if buffers are consumed and completed by the device in the FIFO order. Otherwise one will have to separate the management of `avail` ring and `used` ring, allowing buffers to be loaned out and returned independently of their original positions in the queues. The RX direction is not difficult either if one can ensure the virtio device implementation pre-posting buffers early enough before packets arrive so packets can land directly on the buffers. This is what we've done for LKL over RDMA (Figure 8). But it can be difficult if buffer needs to be posted into a separate software component, like our TCP proxy configuration (Figure 3). To avoid the copy there we would have to change the virtio-net driver implementation to allow taking buffers supplied by the device, rather than having the kernel replenishing the buffer ring, which seems difficult to do.

As for the socket calls, literatures are abundant (e.g., [1]) on how to avoid the copy there. We initially thought it would be easier for LKL because there is no issue with the change of protection domains - LKL kernel and users share the same address space and privilege, therefore buffers are readily accessible from both sides without requiring any additional work. Furthermore, LKL employs a no-MMU architecture hence no virtual memory and buffers are always memory resident with no need to lock down their backing pages.

Unfortunately it's not that simple. Although buffer addresses are always valid therefore simple CPU loads/stores always work from both the user and the kernel sides, the main data structure `struct sk_buff` aka `skb` used by the Linux kernel networking code often maintains data segments in an array `frags` involving the `struct page` data structure. The latter is used to keep track of memory at the physical level even on architectures without an MMU. Even if we manage to avoid the use of `frags` and fit the whole large segment in the `skb head` using just the memory address, when the skb hits the IO layer, the kernel will still need to lookup the backing page to perform device IO.

Since the host thread making the socket call is foreign to the LKL kernel, any kernel attempt to look up the backing page on a host supplied buffer address will fail. This implies a set of syscalls requiring such operations, such as `vmsplice(2)` will not work with buffers allocated by the host. One solution (provided by Octavian Purdila) is for the user to use LKL `mmap(2)` with `MAP_ANONYMOUS|MAP_PRIVATE` flag to allocate LKL "kernel memory" to use. This unfortunately requires application changes, and may be awkward since applications often employs `iovec` to compose the protocol headers and payload from different parts of memory.

A separate syscall will need to be invented for the application to know when it's safe to reuse a buffer, i.e., when the LKL kernel TCP stack is done with it. Linux kernel does not currently provide such an interface. We will have to continue to work on it.

For the LKL RX side, zero-copy socket seems easier. One would need to translate the pages from `skb`'s `frags` back to memory address. This is pretty straightforward given LKL's no-MMU and FLATMEM architecture. We will also need a separate syscall for the application to release the `skb` back to the LKL kernel.

Unfortunately more details have to be deferred as we are not able to complete the work originally planned.

## Debugging Facility

There are a vast number of system tools available for the Linux OS. Networking diagnosis tools such as netstat/ss, ethtool, tcpdump, and many others from iproute2, net-tools packages are indispensable for Linux networking engineers to perform debugging, system diagnosis and tuning of a live Linux networking stack effectively.

Since LKL has most of the kernel code linked with it, it supports all the ioctl, netlink socket, /proc, etc interfaces required for these tools to work. Unfortunately, given LKL lives in the user space, it's confined to the single process space where the kernel instance is booted. It seems difficult to bridge calls between a diagnosis tool running in a different process with an LKL instance.

As a starter, we've provided a simple solution to enable access to counters and parameters inside an LKL instance's procfs, sysfs,..., etc. Upon receiving a specific `signal(7)`, the process running LKL will spawn a thread to provide a simple command line interface to the LKL console, allowing one to mount special filesystems like procfs, sysfs,..., etc. Then one can use simple cd, ls, cat, echo, commands to retrieve any counter or set/tune any parameter desired. This simple facility has proven to be extremely useful, allowing us to retrieve the most basic information such as SNMP counters, or turning a simple knob in the stack.

A more general approach to support those tools directly is to have a syscall proxy to pass calls between processes, like the rump sysproxy [3].

## Linker and Loader Issues

We have encountered a number of mysterious failures from the GNU compiler/linker, either during the compiling/static-linkage time or, worse, at runtime when an application is being loaded. Fortunately our intern Andreas Abel[2] has been able to resolve most of the issues for us. The following is one example where we hit a bug that affects LKL used as library code. Applications linked with LKL statically will trigger a segmentation fault during loading, well before execution.

The problem has to do with position independent code (PIC) and `kallsyms`. `kallsyms` adds kernel debug symbols to the `.rodata` section that eventually goes into the `.text` segment. Those symbols reference the address of the `.text` section and require a relocation at runtime when compiled as PIC, which is called `TEXTREL`. When `TEXTREL` exists together with an `ifunc` that produces a type of relocation calling a function in the `.text` section, a segmentation fault may happen. `libatomic` is an example usage of `ifunc` to select the best implementation based on hardware at runtime. When the loader processes `TEXTREL` , it makes the `.text` segment writable temporarily. Since, SELinux, for example, does not allow a segment to be both writable and executable at the same time, the loader sets the permissions of the segment to read/write. However, when it processes the relocation of the `ifunc`, it tries to execute code in the `.text` segment (which is at this point not executable) and segmentation fault is thrown. This issue in the loader is already noticed and patches were proposed but rejected due to the requirement of SELinux [4].

The workaround is quite simple. Just turn off `CONFIG_KALLSYMS`, or move the `kallsyms` data from the `.rodata` section to the writable `.data` section. This issue demonstrates some unexpected challenges in using Linux kernel code as a user library.

## Conclusion

LKL has carried us a long way toward providing a viable user space TCP stack for Google internal use. We still have much work to do; and the following is only a partial list:

- Complete the socket zero-copy work and continue to try to remove remaining copy operations.
- Figure out a less disruptive way to support direct syscall while still keeping all the performance gains.
- More work is needed to further reduce the overhead of the LKL stack from more than 2.5X of host as described previously for Figure 8.
- Attempt the syscall proxy approach to enable more diagnosis tools to run on top of LKL.
- Since LKL does not support SMP, its current performance is capped by how much performance a single CPU core can deliver. One workaround is to shard the application port number space thus allowing multiple LKL instances to run simultaneously. In the long run SMP support seems to be the way to go.

Finally it would be best for the user community if LKL can be accepted into the upstream kernel at some point. We do not have enough knowledge about LKL's past to assess how much obstacle there may be but will work with the maintainer and the rest of the community to achieve this.

## Acknowledgments

Two early members of the team Xiao Jia and Patrick Collins made significant contributions to get our project off the ground locally. We'd also like to thank Andreas Abel for fending off a slew of linker/loader related bugs during his internship with us. Hajime Tazaki, the author of LibOS and now an active contributor of LKL, has provided helpful insights to us. Finally kudos goes to the LKL open source maintainer Octavian Purdila for continuing to improving the codebase.

## References

[1] Chu, J. 1996. Zero-Copy TCP in Solaris. In *Proceedings of the USENIX 1996 Annual Technical Conference*. San Diego, CA: USENIX Association.

[2] Dike, J. 2000. A user-mode port of the linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4*, ALS'00, 7–7. Berkeley, CA, USA: USENIX Association.

[3] Kantee, A. 2011. Kernel servers using rump. `http://www.netbsd.org/docs/rump/sysproxy.html`. [Online; accessed 09-Sep-2016].

[4] 2015. PIE binary with `STT_GNU_IFUNC` symbol and TEXTREL segfaults on `x86_64`. `https://sourceware.org/ml/libc-alpha/2015-08/msg00419.html`. [Online; accessed 30-Aug-2016].

[5] Purdila, O.; Grijincu, L. A.; and Tapus, N. 2010. LKL: The linux kernel library. In *9th RoEduNet IEEE International Conference*, 328–333.

[6] Tazaki, H.; Nakamura, R.; and Sekiya, Y. 2015. Library operating system with mainline linux network stack. netdev0.1.

[2] mail@andreasabel.de