Asymmetric Processing to Reduce Network Jitter

1st Satish Kumar, 2nd Fam Zheng
ByteDance
London, United Kingdom
satish.kumar@bytedance.com, fam.zheng@bytedance.com

Abstract

A RPC request goes through a minimum of four stage network pipeline on the remote server. The NIC DMAs the packet, the kernel receiver stack directs the data to the socket, the application context reads, process and sends the response back, and finally NIC DMAs the data to the wire. The execution of the first and final stage is performed by the NIC in isolation but the other two stages runs in resource sharing environment and constantly interfere the execution of each other. The design is pipelined but the execution is not really pipelined. There is no logical sharing of data between kernel receiver stack and the application as first touches the header of the packet and the later consumes the payload. In this paper, we analyse the benefits of executing the kernel network receiver stack in isolation from the application (sender) context. We benchmark the new isolation execution model using the Redis server and Netpoll RPC framework where throughput increase is 20% and 9% along with reduction in latency by 25% and 15% respectively.

Keywords

Linux Kernel Network, Network Receiver Stack, Network Sender Stack, Event Loops, RPC, SoftIRQ, IRQ, Run to Completion, Pipeline, Redis, Netpoll

Introduction

Following the microservice architecture, the large applications inside the datacenter are broken down into smaller services which communicate with each other using Remote Procedure Calls (RPC). The network traffic generated by service communication (also known as east-west traffic) forms a significant portion of datacenter networking and a server load. As there might be thousands of active connections, these services utilise the epoll event notification subsystem of Linux to multiplex IO inside an event loop. Figure 1 shows the two common variants of the event loop architecture adopted by applications inside the datacenter.

IIn single thread architecture, the task of reading data from socket, processing application logic and then sending out the response is performed in the context of event loop itself. The single thread is used in a non-blocking manner and the IOs are scaled using kernel threads (SoftIRQ for networking). Also, the complexity of multi-threaded concurrency and data ordering is pushed to the kernel threads, simplifying the application flow. This architecture is usually utilised by the applications which are IO bound, for example, Redis server. On the other hand, the multi-threaded architecture is more applicable to CPU bound applications as it creates additional threads to scale application logic. The Netpoll RPC framework is designed following the multi-thread architecture to support application scalability.

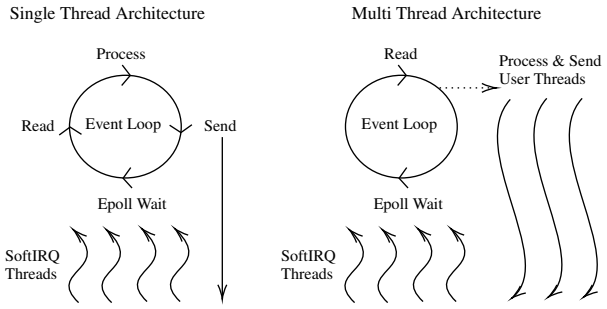

Figure 1: Event loop based application architectures.

Both architectures observe non-blocking single receiver context in the userspace and depend upon kernel threads to scale network IOs. So the systems are usually configured in a scale-out manner by using multiple queues of the NIC, i.e. RSS (Receive Side Scaling), to equally distribute received packets across CPUs. Overall, there are two execution contexts, i.e. IRQ context (consist of NIC IRQ handler and SoftIRQ) and application context. Given the scale out configuration, both the context runs simultaneously on most CPUs, which can be best matched with the Schedulable Pipeline Model from Figure 2.

The Run to Completion model is preferred by simpler networking applications with real time requirements, for ex, L2 and L3 forwarding. The pipeline model or sometimes referred to as graphs provide dedicated resources to every stage of the pipeline, is preferred by more complex applications like deep packet inspection etc. The Linux kernel network stack, runs on the shared CPU resources, is pipeline by design and used by applications for network communication. The pipeline stages are scheduled by the kernel in a concurrent safe implementation. We have referred to this execution model as Schedulable Pipeline.

The jitter, which we define as additional work done by

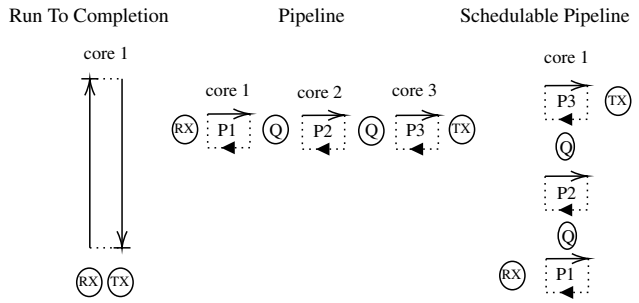

Figure 2: Different network execution models.

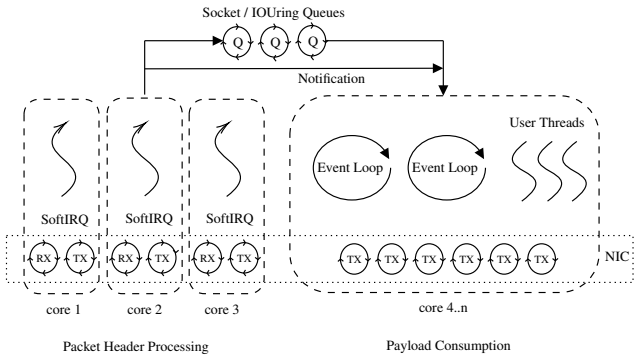

Figure 3: Asymmetric Network Processing (ANP) Model.

the hardware or software apart from processing the request, is minimum on Run To Completion model, intermediate on Pipeline model, and maximum on Schedulable Pipeline model. The reason is quite evident as jitter increases with more complexity.

In this paper, we put forward a new Asymmetric Network Processing (ANP) execution model for Linux kernel where the receiver stack is isolated from the application context to find a better balance between two pipelined models shown above. We analyse our model using the ping pong micro-benchmarks, and present results for Redis and Netpoll RPC benchmarks. In the end, we suggest when and how this model can be applied and additional changes in the kernel to further enhance the benefits of the ANP model.

Asymmetric Network Processing (ANP) Model

The Linux kernel receiver stack is responsible for processing the packet header and gets executed inside the NIC IRQ and SoftIRQ context. At the end of each execution, the IO is inserted into the respective socket queue for the application to consume the data/payload in its own context. There is no logical sharing of data as first operates on the header and later only touches the payload, so it makes sense to separate the receiver stack from the rest of the execution to allow interference free execution of independent logic. Figure 3 presents the Asymmetric Network Processing (ANP) model which isolates the kernel receiver stack to a reserved set of CPUs.

Identifying the number of CPUs needed by the receiver stack is dependent on the workload and can be done dynamically based on policies. In the conclusion and future section, we will talk more about things to consider to define these policies. Once the CPUs are identified to run the receiver stack, it needs to be made sure that single RX and TX pair is assigned to each of them, which can be done by changing the NIC IRQ affinity from procfs. After that, the NIC's receive flow hash indirection table can be modified to redirect all packets equally among the NIC queues which are mapped to the reserved set of CPUs. The ethtool provides the option to modify the hash table. For example, the command "ethttool -X ¡dev¿ equal 4", will configure NIC to redirect all packets to the first 4 RX queues equally. Finally, to create the complete isolation of the receiver stack, task affinity of the user application needs to be changed to the remaining set of CPUs.

The epoll notification subsystem, socket queues and the IO descriptor (skb) should be the only shared data between the receiver stack and the application. The send usually happens in the application context and will be confined to the application CPUs where TX queues are mapped. Point to note that the RX queues are also mapped to the CPUs used by the application (not shown in the figure for simplicity), but they will not generate any traffic due to hash table redirection modification. The TX queues on the reserved CPUs will be used for IP layer communication.

The ANP model thus achieves the full separation of send and receive flow and isolates the receive IRQ contexts from the application context.

Analysis

Except epoll, socket queue and its descriptors, we look for complete isolation of kernel receiver stack with the application context. In this section, we will analyze the behaviour of Linux networking stack with the ANP model.

A ping-pong client-server workload is used for the analysis. The performance numbers and perf events are measured on the server side. There are two different implementations of server, i.e. bm0 and bm1, representing single thread and mult-thread event loop architecture shown in Figure 1 respectively. The server is configured to run four event loops of the same type where connections are equally distributed.

Total number of CPUs allocated for the experiment are 20, each belong to a separate core and part of the socket where NIC is attached. The ANP model (asymmetric) is compared with the configuration where all 20CPUs are shared by the kernel receiver stack and application alike (symmetric). As the ANP model require certain number of CPUs to be reserved for the receiver stack, in this experiment, 8 CPUs are reserved for bm0 and 4 CPUs in the case of bm1 server. The remaining CPUs are used to run the application context, so in the case of bm0 it's a 12-8 split and for bm1 it's a 16-4 split. More reserved CPUs are provided to the single thread architecture server, i.e. bm0, as it consumes higher requests per second than its counterpart bm1, which creates user threads for processing.

The payload size of 1KB is used in all the experiments. The client and server runs on different Intel Xeon machines

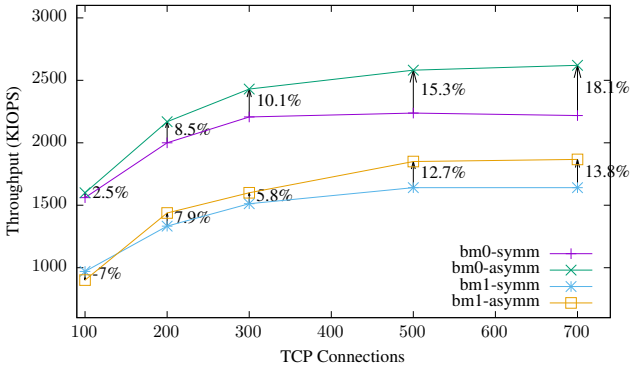

Figure 4: Throughput vs Number of Connections.

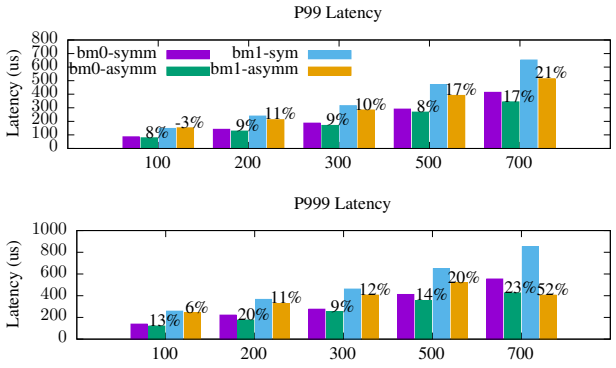

Figure 5: Latency vs Number of Connections.

with 25gbps NIC connected to each other via TOR switch. The kernel version used is v6.7.

Graphs in Figure 4 and 5 provide throughput and latency comparisons of the ping-pong benchmarks. The number of connections varies on the X-axis. Comparing the two models, the performance characteristics achieved by the ANP model becomes more significant with higher number of connections. Higher connections generate high network load, thus increasing the jitter in the system that ANP model is able to reduce. For bm0 and bm1 servers, the throughput achieved is 2-18% greater, and the 99 and 99.9 percentile latencies are reduced by 6-50%.

The Topdown analysis from perf was used to understand the results and differences. Table 1 pesents the TopdownL1 metrics in different experiments. To make sure that the metric numbers are comparable, the benchmarking was performed by generating a fixed number of IOs. As the amount of work generated is constant, the number of instructions executed is fairly the same (0.02% and 4% less for ANP). The Topdown metrics suggest that the ANP model reduces the overall bottleneck on frontend as well as on the backend, which is leading to better retiring instructions and IPC (instructions per cycle) numbers.

Further analysis of backend and frontend perf events reveals a pattern. Table 2 shows the percentage difference between two models, where a negative value means ANP

	bm0-symm	bm0-asymm	bm1-symm	bm1-asymm
instructions	3972234172663	3971267766582	5376436857531	5140460280980
inst per cycle	1.25	1.32	1.08	1.14
tma_backend_bound	47.8	46.2	50.1	50
tma_bad_speculation	2.5	2.6	4.4	4.3
tma_frontend_bound	23.2	23.3	22.6	21.7
tma_retiring	26.4	27.9	22.9	24.1

Table 1: TopdownL1 metrics.

	bm0	bm1
MEM_INST_RETIRED.ANY	0.3%	4.7%
CYCLE_ACTIVITY.STALLS_MEM_ANY	2.8%	7.9%
EXE_ACTIVITY.BOUND_ON_STORES	15.4%	36.1%
CYCLE_ACTIVITY.STALLS_L1D_MISS	-5.6%	5%
CYCLE_ACTIVITY.STALLS_L2_MISS	-5.3%	5%
CYCLE_ACTIVITY.STALLS_L3_MISS	94.6%	91.1%
ICACHE_TAG.STALLS	33.3%	24.7%
ICACHE_DATA.STALLS	14.8%	29.8%

Table 2: Cache hierarchy perf miss events.

model had higher value for that particular counter. The retired memory instructions are fairly the same. The L1 and L2 caches are local to the core and L3 is shared. Apart from L1D and L2 cache, all the other caches stalls are reduced inside ANP model. Specifically, the frontend caches, which consist of TLB (ICACHE_TAG) and L1 instruction cache (ICACHE_DATA), where the stalls are significantly reduced, its mainly because caches are more aligned and friendly to the type of load running on that CPU after isolation. For bm0, the slight increase in L1D and L2 stalls, and decrease in L3 stalls signify an increase in cross core communication. This is expected as ANP will have slightly more cross core communication after the separation. For bm1, which creates additional user threads, cross core communication is hidden by a better hit ratio of application threads.

To better understand the data sharing pattern, the L2 stalls are sampled to pin point the code paths. Table 3 shows the list of functions where large amount of stalls ocurred, and as before, the negative value means a higher value for ANP model. The function _copy_to_iter, copies the data to the userspace and its close to zero delta confirms that there is no logical sharing of data between kernel receiver and application context. Functions like sock_poll, tcp_poll, do_epoll_wait and _raw_read_lock_irqsave are all part of the epoll notification system where cross core communication is expected. The tcp_queue_rcv function inserts the entry into the socket queue. However, further investigation is needed on other code paths and the sharing of skb data structure.

In general, the analysis shows that the efficiency of caches is one of the main reason behind better performance of the ANP model. The other obvious reason being the interference free execution of independent contexts to reduce resource contention. It should be noted that the analysis presented in this section is not complete as there are various other scenarios where new data flows may be revealed. For example, in our experiments, pfifo queuing discipline is used but the flow may change with a rate limiting traffic control algorithm.

	bm0
tcp_ack	-53.1%
_copy_to_iter	1.5%
sock_poll	-23.8%
skb_release_data	-42.1%
tcp_recvmsg_locked	-12.1%
__check_object_size	-1.9%
tcp_queue_rcv	-72.7%
tcp_poll	-29.3%
tcp_check_space	-77.2%
skb_attempt_defer_free	9.4%
_raw_read_lock_irqsave	-103.7%
tcp_rcv_established	-65.1%
__list_del_entry_valid_or_report	-5.6%
__inet_lookup_established	-120.4%
do_epoll_wait	-97.6%
napi_pp_put_page	-58.6%
__lock_text_start	-72.2%
native_queued_spin_lock_slowpath	-569.9%

Table 3: CYCLE_ACTIVITY.STALLS_L2_MISS event sampling.

Results

In this section, the results of Redis server and Netpoll RPC framework with the ANP model are presented. The experimental setup remains the same as the previous section, and the same terminologies are followed.

Commonly used Memtier benchmark is used to generate load on the Redis cluster. The benchmark is configured to create 240 connections over 30 client threads which generates a random value size workload (capped at 1KB) in the 2:1 ratio of get and set commands. The same memtier configuration is used in all the cases and the number of servers in the Redis cluster are increased along with the total available CPUs. Following configurations are benchmarked, in which, the number of redis servers are 4, 8, 12, 16 and 20, and the total CPUs allocated are 12, 20, 28, 32 and 32 respectively. In the case of ANP, the number of reserved CPUs used for the receiver stack, following the same order as the previous line, are 4, 8, 8, 8 and 8, thus creating a split of 8-4, 12-8, 20-8, 26-8 and 26-8 respectively.

Netpoll is written in Golang and by default it creates one event loop per 20 go processes. In our experiments, the available CPUs are fixed to 16, so Netpoll only creates one event loop thread context. A total of four CPUs are reserved for the ANP model, creating a 12-4 split.

Figure 6 shows the throughput graphs for Redis-Memtier and Netpoll benchmarks. The ANP model provides better throughput in all the cases of Redis and Netpoll, ranging between 4-22% and 5-8% respectively. Figure 7 presents the 99 and 99.9 percentile latency graphs. The P99 latency in Redis

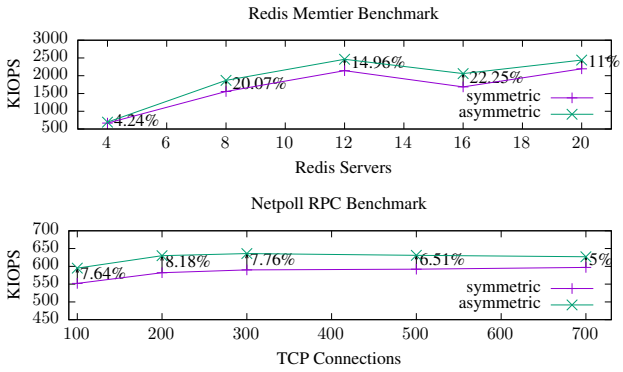

Figure 6: Throughput comparison of Redis and Netpoll.

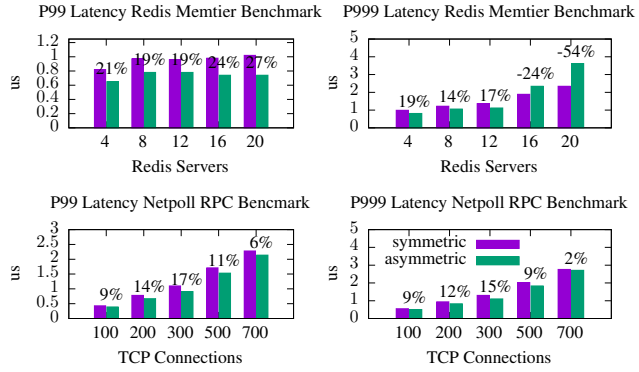

Figure 7: Latency comparison of Redis and Netpoll.

is reduced significantly by 19-20% in all the cases but when the number of Redis servers is increased to 16 and 20, there is an increase of P999 latency by -24% and -50%. The less available concurrency for redis servers in these two scenarios, after ANP model separation, is the reason for high tail latency. We will talk more in the next section about when it's best to use ANP model. For Netpoll, the P99 and P99 latencies are reduced in all cases by 6-17% and 2-15% respectively.

Conclusion

In this paper, we put forward the new ANP model which provides promising results of removing jitter in the shared system. The analysis showed that a better cache efficiency was achieved and, importantly, the ANP model achieves an interference free execution environment for RX and TX flows. The cross core communication after isolation was identified to be not significant in the Linux kernel.

The ANP model cannot be applied in all scenarios and points below are some considerations:

- ANP model reduces the concurrency of the application, so on a highly loaded system it may have side effects specific to the application. For example, Redis server showed high tail latency when number of servers increased.
- The model makes sense when there is a one-to-many relationship between application receiver and kernel softirq

threads. If the application follows a blocking design which creates a thread per connection, then RFS (Receiver Flow Steering) and other strategies may prove better.

Future work on Linux kernel:

- As stated before, the analysis is not yet complete and requires more experimentation with other common configurations.
- More investigation on data sharing blocks is needed so that any unnecessary cross core communication can be avoided.
- On the last note, the integration of io_uring with SoftIRQ context can provide significant advantages to Netpoll like architecture. As in these applications, the thread which reads the data and the context which consumes the data is separate. SoftIRQ copying the data directly to the userspace buffer will remove overhead from the application event loop.

References

[1] Netpoll CloudWeGo Github https://github.com/cloudwego/netpoll.

[3] Redis In Memory Database https://github.com/redis/redis.

[3] Memtier: Redis Open Source Benchmark https://github.com/RedisLabs/memtier_benchmark.

[4] Hesam Zolfaghari; Haseeb Mustafa; Jari Nurmi Run-to-Completion versus Pipelined: The Case of 100 Gbps Packet Parsing 2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR). https://ieeexplore.ieee.org/document/9481797.

[5] Elder Vicente; Rivalino Matias Jr. Exploratory Study on the Linux OS Jitter 2012 Brazilian Symposium on Computing System Engineering https://ieeexplore.ieee.org/document/9481797.