Userspace Networking in gem5

Johnson Umeike Siddharth Agarwal[§] Nikita Lazarev[†] Mohammad Alian
[§]University of Illinois Urbana Champagin [†]MIT, CSAIL University of Kansas

***Abstract*—Full-system simulation of computer systems is critical for capturing the complex interplay between various hardware and software components in future systems. Modeling the network subsystem is indispensable for the fidelity of full-system simulations due to the increasing importance of scale-out systems. Over the last decade, the network software stack has undergone major changes, with userspace networking stacks and data-plane networks rapidly replacing the conventional kernel network stack. Nevertheless, the current state-of-the-art architectural simulator, gem5, still employs kernel networking, which precludes realistic network application scenarios.**

In this work, we first demonstrate the limitations of gem5's current network stack in achieving high network bandwidth. Then, we enable a userspace networking stack on gem5. We extend gem5's NIC hardware model and device driver to support userspace device drivers running the DPDK framework. Additionally, we implement a network load generator hardware model in gem5 to generate various traffic patterns and perform per-packet timestamp and latency measurements without introducing packet loss. We develop a suite of six network-intensive benchmarks for stress testing the host network stack. These applications, based on DPDK, can run on both gem5 and real systems. Our experimental results show that enabling userspace networking improves gem5's network bandwidth by 6.3× compared with the current Linux kernel software stack. We characterize the performance of DPDK benchmarks running on both a real system and gem5, and evaluate the sensitivity of the applications to various system and microarchitecture parameters. This work marks the first step in refactoring the networking subsystem in gem5.

I. Introduction

The evolution of networking technology enabled hundreds of gigabits per second inter-server data transmission rates in datacenters, and terabit per second network interfaces are on the horizon [1]–[3]. Such advances in the network hardware performance ignited the research and development effort to re-architect the software stack to deliver genuine hardware performance to the applications. Over the past decade, userspace and data-plane networking technologies have emerged as replacements for the kernel network stack to minimize its associated overhead on network bandwidth and latency [4]–[9]. The Data Plane Development Kit (DPDK) [10] is an open-source software project that provides a set of data-plane libraries to move packet processing from the kernel to processes running in userspace. Applications developed using DPDK enjoy higher network performance as the library removes userspace/kernel context switches, uses huge pages for network buffer allocation, implements run-to-completion application processing, and utilizes a polling mode driver for interacting with the NIC.

As the network bandwidth approaches the local memory bandwidth of end hosts, proper handling of network rates in the processor microarchitecture and memory hierarchy is essential to deliver high-quality end-to-end performance for emerging exascale applications. Architectural simulation has long been the primary tool for the early evaluation of future computer systems. However, current architectural simulators fall short when it comes to modeling the state-of-the-art network hardware and software stack. For example, gem5 [11], [12] only supports kernel networking and only delivers ∼10Gbps network bandwidth running the iPerf TCP throughput test. FireSim is an FPGA-based cycle-accurate simulator, which also only supports kernel networking and delivers ∼1Gbps iPerf TCP bandwidth [13]. With such low network bandwidth, architectural simulation cannot be used to study the implications of future terabit-per-second NICs on core micro-architecture and memory hierarchy.

We identify three shortcomings in current architectural simulators with respect to evaluating future networked systems:

- Existing simulators have outdated models for the networking subsystem and can deliver at most tens of gigabits per second network data rates to the processor and memory hierarchy.
- Existing simulators use load-generator applications to stress the node-under-test. However, when dealing with high network rates, the load-generator applications, annotated with performance sampling functions, may emerge as the end-to-end bottleneck. This bottleneck can lead to measurement errors and hinder users from effectively saturating the node-under-test. Additionally, the load-generator node has the potential to impede the overall simulation speed.
- There is no networking benchmark suite tailored for running on simulators with standard metrics and evaluation methodology.

In this work, we fill these gaps by enabling DPDK on the gem5 simulator, extending gem5 with a load-generator hardware model, and implementing a suite of five network-intensive applications for stress testing end-host networking subsystem. DPDK on gem5 bypasses the Linux kernel and delivers the maximum network bandwidth that a given processor architecture and memory hierarchy can sustain. In other words, we enable full-system gem5 to simulate networked systems in which the network stack is no longer the bottleneck; instead, in our setup, the processor, interconnect, or the memory are the bottlenecks in network packet processing [14]. We enable gem5 to run unmodified DPDK applications.

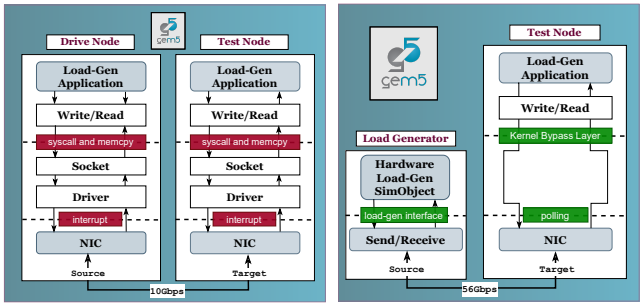
We introduce a network load generator hardware model that can be used to inject packets to the simulated network with

configurable rate, packet size, and traffic pattern. The load generator can generate synthetic traffic or replay pcap traces captured on a real system using a packet capture tool like tcpdump [15] or dpdk-pdump [16]. Hardware load generators are widely used in the industry to stress test the network subsystem without introducing any packet loss.

We execute DPDK applications on both gem5 and an ARM Neoverse N1 server platform, aiming to characterize the variations in network performance between gem5 and the real system. Then, we show that at large packet sizes, gem5's DMA engine is the bottleneck. Furthermore, we leverage gem5 to analyze the sensitivity of network-intensive applications to microarchitectural configurations. Lastly, we utilize the load generator hardware model to load a Memcached server, using kernel network stack as well as DPDK.

II. Background and Motivation

A. Network Software Stacks



(a) Current gem5 (b) This work

Fig. 1. (a) Baseline dual-mode gem5 with Linux kernel software stack for evaluating networked systems, (b) userspace networking in gem5 with hardware load generator simulation model.

Network packet processing in the Linux kernel is hindered by frequent system calls and context switches. Additionally, there are frequent buffer copies within the kernel software stack and between kernel and userspace buffers. These overheads are compounded by the extended latency associated with interrupt processing and notification [2], [14]. Userspace software stacks address these overhead by offloading packet processing to userspace processes. They utilize large buffer allocations or huge pages, implement zero-copy data transfer [17], apply thread-to-core pinning, adopt run-to-completion application implementations, and incorporate polling for receive (RX) and transmit (TX) completion notifications.

DPDK [10], first released by Intel in 2010, comprises a set of data-plane libraries that enable direct control and access to the NIC hardware from a userspace application. DPDK reserves pinned huge pages and allows the NIC to DMA packet data directly into the application's buffers. Furthermore, DPDK employs a polling mode driver to eliminate the overheads of interrupt processing, which most often arise from frequent context switches. DPDK application can be implemented in two modes:

Run to completion mode: where the packet processing loop is: (1) retrieve RX packets through Polling Mode Driver (PMD) RX API, (2) process packets on the same logical core, (3) send pending packets through the PMD TX API.

Pipeline mode: where the cores pass packets between each other via a user-level ring buffer for efficient packet processing.

B. Current gem5 Network Stack

The current NIC simulation object in gem5 loosely models the Intel 8254x NIC series [18]. Fig.1a illustrates how gem5 currently simulates a two-node system connected to each other using a direct Ethernet link. The goal is to have a *Test Node* that runs a network application (e.g., forwarding, key-value store, etc.) receiving network requests from a *Drive Node*. These nodes can be simulated using a single gem5 process, also referred to as dual-mode gem5 (as shown in Fig.1a), or using dist-gem5 [19], which runs two separate gem5 processes. Each process simulates either Test or Drive nodes in parallel, synchronizing them at every minimum simulated network latency. The *Drive Node* runs a load generator application that establishes a connection with the application running on the *Test Node* and sends requests at various load levels, sizes, and inter-arrival time distributions. Because optimizing the performance of load generator applications is often not the focus, the *Drive Node* does not need to be simulated with high fidelity, and a functionally correct simulation is usually sufficient.

Default gem5 uses a Kernel network stack that can only sustain ∼10Gbps network bandwidth using beefy O3 (Out-of-Order) ARM cores running at 3GHz frequency. See Sec.VI for detailed experimental methodology. Such low network bandwidth does not sufficiently exercise the hardware and software stacks. Hence, the current gem5 network subsystem is not useful for studying systems that support hundreds of gigabits per second network throughput.

C. Hardware Traffic Generators

One of the main concerns when evaluating networked systems is loading the *Test Node* with real traffic to measure network bandwidth and per-packet, round-trip network latency without introducing extra latency or packet drop at the load generator node (i.e., *Drive Node*). The industry standard practice involves using hardware traffic generators equipped with FPGA line cards [20], [21] to generate packets with configurable traffic patterns, sizes, and protocols, while also providing detailed network statistics for each transmitted and received packet [22]. In this work, we enhance gem5 with a hardware load generator that replaces the *Drive Node* (as shown in Fig.1b).

III. Linux Kernel Bypass in gem5

This section discusses the changes we made to gem5 and the DPDK framework to enable userspace networking and implement the hardware load generator model in gem5. We do not make any changes in the Linux kernel.

offset	bits[31:24]	bits[23:16]	bits[15:8]	bits[7:0]
0x00	Device ID		Vendor ID	
0x04	Status		Command	

Fig. 2. First 8 bytes of the PCI configuration space that includes PCI Command Register.

A. Changes to gem5

The changes in gem5 are limited to the PCI model to enable userspace I/O (UIO) driver and the NIC model to allow byte granular PCI configuration space accesses.

1) Enable userspace I/O Driver: `uio_pci_generic` driver in Linux enables a userspace application to directly access the address space of a PCI device. DPDK uses this driver to enable the userspace application to access the PCI configuration space and implement a Polling Mode Driver (PMD). Mainline gem5 does not enable the `uio_pci_generic` driver during boot as the PCI Command Register is not fully implemented in gem5. Fig.2 shows the first 8 bytes of the PCI configuration space that includes the 16-bit Command Register at offset `0x04`. The baseline gem5 implements bits 0-9 of the Command Register but does not implement bit-10, which is the *interrupt disable* bit. We implement *interrupt disable* bit in gem5 PCI model, so the Linux kernel can disable the interrupts for the PCI devices on gem5, which is necessary to support `uio_pci_generic` driver.

2) Enable Byte-Granular Access to PCI Configuration Space: gem5 only supports 16-bit accesses to the Command Register shown in Fig.2. In fact, this is rational since the size of the Command register is 16 bits. However, we observed that DPDK accesses the Command register using 8-bit memory accesses. Such byte-granular accesses are being ignored in gem5, and therefore DPDK cannot properly read and write the upper half of the Command Register (offset `0x05` of the PCI config space). We extended the `readConfig` and `writeConfig` functions in the gem5 PCI model to enable byte-granular accesses to the Command Register.

3) Enable the NIC Model to Correctly Operate with a PMD: NIC devices keep a handful of available descriptors (usually 32 to 64 descriptors) that can be populated upon receiving a packet on an on-chip cache which is called *descriptor cache*. The descriptor cache improves the performance as the NIC does not need to fetch available descriptors from the CPU memory on demand. The NIC gradually writes back the descriptor cache to the CPU memory (using DMA), and then the CPU is notified of received packets.

gem5's NIC model writes back the received descriptors based on a threshold set by the Linux kernel. Once the number of used descriptors exceeds a threshold, the NIC initiates a writeback. When using a Polling Mode Driver (PMD), the threshold registers in the NIC model are not properly set, and thus the NIC starts writing back the descriptors when all of them are used. This means that packets are DMAed to the CPU memory in large batches (32 to 64 packets), which causes unrealistic pressure on the CPU memory subsystem and increases the possibility of packet drops at high receive rates. We implemented a parameter for the NIC where the user can control the threshold of descriptor writebacks in gem5.

4) Direct Cache Access: We noticed that DPDK performance of baseline gem5 that models a conventional system without Direct Cache Access support is substantially lower than that of real hardware (§VII). We leverage prior work implementation to enable Direct Cache Access (DCA) in gem5 [23]. ARM refers to the DCA implementation as Cache Stashing technology.

gem5 connects the I/O bus [1] directly to the memory controller. Enabling DCA in gem5 requires connecting the I/O bus to the Last Level Cache (LLC), partitioning LLC ways between DCA ways and core ways, and flagging I/O memory requests to distinguish them from core memory requests. We follow the implementation of prior work [23].

5) Implement Interrupt Mask Register in the NIC model: The last modification to gem5 is to implement Interrupt Mask Register in the `i8254xGBe` device model. Interestingly, this register is included in the `i8254xGBe` model, but the read and write methods for accessing the register are not implemented in the current gem5 release. We implemented the read and write methods to enable DPDK to launch its PMD.

B. Changes to DPDK

The DPDK Environment Abstraction Layer (EAL) relies on vendor ID checks to match a device and a PMD. We modify the DPDK source to skip these checks and force the matching of the gem5 device to NIC model PMD. Unmodified DPDK cannot fetch the correct vendor ID when running on gem5 and therefore fails to call the proper PMD. We suspect this is because some manufacturer-specific information is missing in the gem5 NIC model. Note that skipping the vendor ID test does not adversely impact the gem5 simulations, as the current gem5 release has only an e1000 NIC model. If new NIC models are added to gem5, the DPDK framework should be recompiled after hard-coding the PMD to use a different NIC model.

IV. Hardware Load Generator Model

The hardware load generator model can generate packets at arbitrary rates, sizes, and traffic patterns. We implement a simulation object called `EtherLoadGen` that has a single Ethernet port and can directly connect to the NIC port of a simulated node, as shown in Fig.1b. Therefore, for simple network benchmarking, there is no need to run distributed or dual-mode gem5 simulations. Instead, one can simulate only a test node that is directly connected to the `EtherLoadGen`. This approach enhances both simulation speed and the accuracy of simulations, as `EtherLoadGen` does not introduce client-side queuing. Also, `EtherLoadGen` can be used to easily model open or closed loop clients [24].

`EtherLoadGen` can operate in two modes: *synthetic* mode and *trace* mode. In synthetic mode, `EtherLoadGen` sends

[1] I/O bus models PCIe bus in a real system

packets based on a set of configurable parameters such as packet rate, packet inter-arrival time distribution, packet size, and protocol. In trace mode, `EtherLoadGen` replays a trace that is collected from a real system, e.g., using tcpdump [15]. `EtherLoadGen` can either use timestamps in the trace to send packets or override it by the packet inter-arrival time distribution.

The `EtherLoadGen` trace mode is based on the standard Packet CAPture (PCAP) files which can be generated and analyzed by, for example, tcpdump/wireshark from real traffic. The load generator parses PCAP files via the Linux PCAP library [25] and reads the networking trace for each packet. It then modifies the destination physical address in the packet's Ethernet header to match the one in the simulated system. The modified packet is dispatched to the simulated NIC at either a statically configured rate or based on the timestamp information from the original trace if the real-traffic throughput needs to be reproduced. Note that userspace networking traffic generally cannot be captured with tcpdump or any other OS-based utility. We use the dpdk-pdump application [16] to record `.pcap` traces for the DPDK applications. This approach is generalizable to any DPDK system and is compatible with our `EtherLoadGen`. In order to replay DPDK traces, we integrate PCAP file generator into our DPDK-based KVS client directly by using dpdk-pdump library [26].

In the synthetic mode, `EtherLoadGen` creates Ethernet packets with the specified size and sends them at a fixed rate to the Ethernet port. The synthetic protocol that we support for now is plain Ethernet packets. Connection-less protocols such as UDP can be supported with minimal effort. Trace mode is a very convenient way to generate load for more complex protocols such as key-value stores or HTTP. In synthetic mode, `EtherLoadGen` adds a timestamp to each outgoing packet at a configurable offset and compares the timestamp with the current tick on incoming packets to compute per-packet round-trip latency. `EtherLoadGen` reports mean, median, standard deviation, and tail latency of network packets in the statistics file. It also produces a packet drop percentage and a histogram of packet forwarding latency. The load generator model enables simple network benchmarking in gem5 without the need to simulate multiple system nodes. This is similar to the practice in the industry for using hardware load generators to evaluate the performance of the network [20].

`EtherLoadGen` also supports a bandwidth test mode where it gradually increases the bandwidth to find the maximum sustainable bandwidth of a server, which is the bandwidth at the knee of the bandwidth vs. packet drop graph (or bandwidth vs. latency graph). Unlike the Garnet load generator that is used in gem5 to inject traffic to the network on chip [27], `EtherLoadGen` connects to NIC ports and is used to generate Ethernet network traffic.

V. Benchmark Suite

We introduce six networking applications, four of which are network-intensive microbenchmarks and two real in-memory key-value stores. These are used to benchmark the end-host performance in processing network packets. We enable `EtherLoadGen` to load these applications when they are run on the Test Node.

`TestPMD` is the unmodified testpmd DPDK application [28] that implements packet RX and TX functionality with configurable forwarding modes. For example, `TestPMD` can receive packets from NIC in configurable batch sizes, swap their source and destination MAC addresses (if macswap forwarding mode is enabled), and then enqueue them in the TX ring buffer for transmission. `TestPMD` is a shallow network function, meaning that it only uses the L2 header (14 bytes – 12 bytes for source and destination MAC addresses and 2 bytes for the frame length) to make the forwarding decision.

`TouchFwd` is a DPDK application that forwards received packets at L2 layer while touching the entire payload. In other words, `TouchFwd` extends `TestPMD` with an extra loop that brings the payload to the core (subsequently to L2 and L1 caches). `TouchFwd` can be used to model deep network functions such as Deep Packet Inspection [29].

`TouchDrop` is a DPDK application that receives packets from NIC, touches the entire header and payload, and drops it. `TouchDrop` is a variation of `TouchFwd` that does not implement the transmission phase. `TouchDrop` can be used to evaluate the performance of end-host packet reception.

`RXpTX` is a configurable micro benchmark that implements three phases in a run-to-completion DPDK application. `RXpTX` receives a burst of packets from NIC, waits for a *processing* interval, and transmits them over the network. Changing processing time can model network functions with different DMA to core use distances. `RXpTX` can be used to evaluate the performance of various policies for Direct Cache Access (DCA) [30].

`MemcachedDPDK` is a simple in-memory key-value store implemented on top of DPDK and thus achieves higher throughput and lower latency per request. We implement packet trace collection from a real system using dpdk-pdump, enabling their subsequent replay by the `EtherLoadGen` model in gem5.

`MemcachedKernel` is an in-memory key-value store implemented using the memcached [31] library and Linux POSIX APIs. We have enabled `EtherLoadGen` to send GET and SET requests to the memcached server, with configurable sizes for keys and values. Currently, `EtherLoadGen` only supports replaying memcached traces for UDP connections because adding support for TCP would require implementing a TCP state machine inside `EtherLoadGen` (which is a future work). We run the memcached client on a real system and collect UDP traces for the warm-up and run phases of the application using tcpdump. `EtherLoadGen` replays the tcpdump trace to emulate the client process and load the memcached server running on the Test Node. `MemcachedKernel` is not a DPDK application, we provide it for performance comparison of DPDK and kernel network

stacks.

VI. Methodology

A. Experimental Setup

Table I. Simulated and real system configurations.

Parameters	gem5	Ampere Altra
Core freq:	3GHz	3GHz
Superscalar	4 ways	4 ways
ROB/IQ entries	128/120	128/120
LQ/SQ entries	68/72	68/72
Int & FP physical registers	256 & 256	unknown
Branch predictor	BiModeBP	unknown
BTB entries	8192	6000
L1I/L1D (size, assoc)	64KB,4/64KB,4	64KB,4/64KB,4
L2 (size, assoc)	1MB,8 ways	1MB,8 ways
L1I/L1D/L2 latency	1/2/12	unknown
L1I/L1D/L2 MSHRs	2/6/16	unknown
DRAM/mem size	DDR4_2400/64GB	DDR4_3200/255GB
DCA/DDIO [32]	default enabled	disabled [33]
Network latency	200μs	200μs (ping RTT)
Network Bandwidth	100Gbps	100Gbps
DPDK Version	20.11.3	21.11.0
Operating system	Linux kernel 5.15.79	Linux kernel 5.15.0
gem5 version	v21.1.0.2	N/A

We use an Ampere Altra Max server equipped with an ARM Neoverse N1 CPU and an NVIDIA ConnectX-6 NIC to compare the performance and microarchitectural statistics of our simulated Test Node with a real system setup. For the Drive Node, we use a Dell server equipped with an Intel Xeon Scalable Gold 6242 CPUs and an NVIDIA connectX-6 NIC running Pktgen [34]. Table I shows the gem5 configuration we used for the experiments that loosely models Ampere `altra`.

To build a kernel and disk image for gem5, we use the Buildroot tool [35]. The kernel needs to be compiled with support for huge pages, and the kernel module `uio_pci_generic`. Listing 1 shows the kernel configuration option needed to be enabled in buildroot tool for DPDK.

```
CONFIG_HUGETLBFS=y
CONFIG_HUGETLB_PAGE=y
CONFIG_UIO=y
CONFIG_PCI=y
CONFIG_UIO_PCI_GENERIC=m
```

Listing 1. DPDK Kernel CONFIG options.

```
1 modprobe uio_pci_generic
2 dpdk-devbind.py -b uio_pci_generic 00:02.0
3 echo 2048 > /sys/kernel/mm/hugepages/hugepages-2048
     kB/nr_hugepages
4 dpdk-testpmd -l 0-3 -n 4 -- --nb-cores=1 --forward-
     mode=rxptx --proc_times=10
```

Listing 2. Bash script for setting up the environment for running DPDK applications on gem5.

Listing 2 shows the bash script for enabling the Userspace IO (UIO) driver (line 1), binding it to a NIC port (line 2), allocating huge pages, and lastly, starting the `RXpTX` application. As shown in the listing, the procedure for running DPDK applications on gem5 is identical to running DPDK apps on bare metal hardware.

For gem5 simulations, we sufficiently warm up the Test Node's microarchitectural states (e.g., caches, and BTB) prior to collecting simulation statistics. We set the warm-up period to 200 ms, as this experimentally shows a stable simulation environment.

To demonstrate the effectiveness of the DPDK software stack on gem5, we executed the applications listed in Sec.V with varying packet sizes and configurations. For `MemcachedKernel` and `MemcachedDPDK`, our Memcached client implementation generates key and value sizes using a Zipfian distribution with control parameters for key length/value length, specifically: min = 10, max = 100, and skew = 0.5. The payload is encapsulated in a Memcached UDP header, a request header containing metadata, and an Ethernet II frame header. Since our focus in this work is on the performance of the network software stack, we do not populate the Memcached server with a large number of key-value pairs. We warm up the Memcached server with 5000 keys and configure `EtherLoadGen` to load the server with 10,000 SET and GET requests during the simulation phase. We set the ratio of GET/SET to 80%. To keep track of per-request latency, the hardware `EtherLoadGen` model tracks a map of outstanding requests using the request ID field in the Memcached request packet.

B. Collection of Real System Metrics

Considering the inherent noise in real system experiments, ensuring result accuracy becomes crucial. In each experiment, we conduct a minimum of seven samples for each metric. This quantity was determined through initial experiments that demonstrated its sufficiency in keeping the margin of error within 5% of the average result. The reported result is the average of these samples.

VII. Results

In this section, we present the performance and microarchitectural statistics of running the applications discussed in Sec.V on gem5 and the real system setup. We evaluated all the benchmarks in the results section except for `TouchDrop` because we could not define the maximum sustainable bandwidth (MSB) metric that we used for evaluating other benchmarks, as the drop rate of `TouchDrop` is always 100%. Our primary goal is to address the following questions: How accurately does gem5 model the network performance of a state-of-the-art ARM server? What are the sources of errors in gem5 network modeling? How does the sensitivity of a userspace network stack differ from that of a kernel network stack in response to micro-architectural configurations? Additionally, what are the use cases for enabling userspace networking in gem5?

A. Analysis of Packet Drops

To understand and analyze the experimental results, it is important to develop a good understanding of the life cycle of a packet in a run-to-completion DPDK application. The diagram in Fig.3 illustrates the life cycle of a packet in a run-to-completion DPDK application. As soon as a packet is received, the NIC enqueues it in an on-chip SRAM buffer referred to as RX FIFO. The DMA engine in the NIC transfers

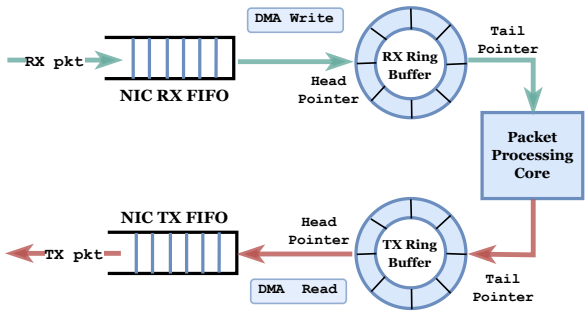


Fig. 3. Packet life cycle in a run-to-completion DPDK application.

RX packets from the RX FIFO to an RX Ring buffer. The NIC caches several RX Ring descriptors and replenishes the descriptor cache when the number of cached descriptors drops below a configurable threshold. If the DMA engine cannot replenish the descriptor cache and deplete NIC RX FIFO, then the NIC RX FIFO becomes full, and we experience packet drops. We identify such packet drops as *DmaDrops*.

Once packets are in the RX Ring Buffer, the DPDK's polling mode driver (PMD) detects them and reads them to the core for processing. If the core is slow in processing the RX packets, then the RX Ring Buffer becomes full, which halts the NIC DMA engine, and soon the NIC RX FIFO becomes full, and we experience packet drops. We identify such drops as *CoreDrops*.

Lastly, once the core completes the packet processing, the application often requires sending a response back to the sending node. That includes setting the TX descriptor pointer in the TX Ring Buffer to point to the transmitting data. We consider memory copies from application buffers to DMA buffers to be part of the packet processing on the core. After the TX descriptors are set, DPDK configures the DMA engine of the NIC to read the enqueued packets from the TX Ring Buffer and transmit them over the Ethernet wire. If DMA read phase from TX Ring Buffer cannot keep up with the rate at which the core pushes packets in, then TX Ring Buffer becomes full, which in turn stalls the core from processing incoming packets. Soon after, the RX Ring Buffer becomes full, which eventually results in packet drops at the NIC. We identify such packet drops as *TxDrops*.

Figure 4 shows the finite-state machine (FSM) we designed to classify different types of network packet drops in gem5. A three-bit number represents each state. If the leftmost bit is 1, NIC RX FIFO is full, and we drop packets. If the middle bit is 1, the RX Ring Buffer is full; if the right-most bit is 1, the TX Ring Buffer is full. We transition between states on packet reception and make transition decisions based on the state of RX FIFO, RX Ring Buffer, and TX Ring buffers. The initial state is 0, 0, 0, indicating a balanced system where none of the buffers are full. The FSM tracks the previous state of buffers and increments the DmaDrop, CoreDrop, and TxDrop counters.

In Fig.5, we present a breakdown of packet drops during the execution of `MemcachedKernel`, `MemcachedDPDK`, `RXpTX`, `TouchFwd`, and `TestPMD` in gem5. We set the

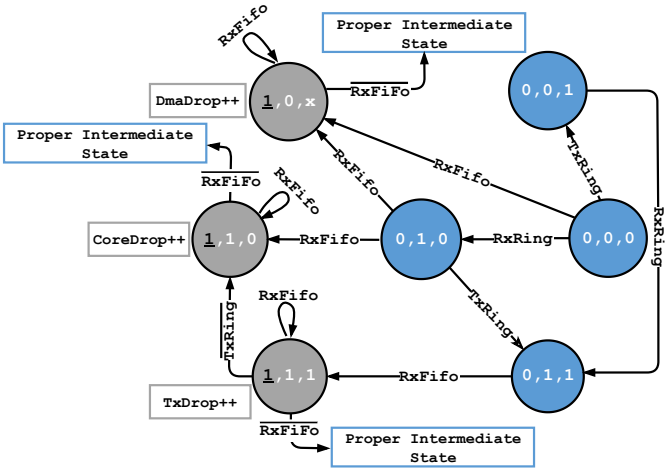


Fig. 4. FSM for identifying the cause of a network packet drop. Each state is identified by a three-bit binary number that encodes the following events: {NIC RX FIFO Full, RX Ring Buffer Full, TX Ring Buffer Full}. The blue-colored states are the intermediate states that one or both of the Ring buffers are full but there is no packet drop. The gray-colored states are the states in which we experience packet drops. The transitions between states take place at each packet RX. When at a gray-colored state and RxFifo is no longer full, then on the next RX packet, we transition to a proper intermediate state. "x" is "don't care."

network bandwidth to the knee of the bandwidth vs. packet drop rate curve, where we start seeing packet drops.

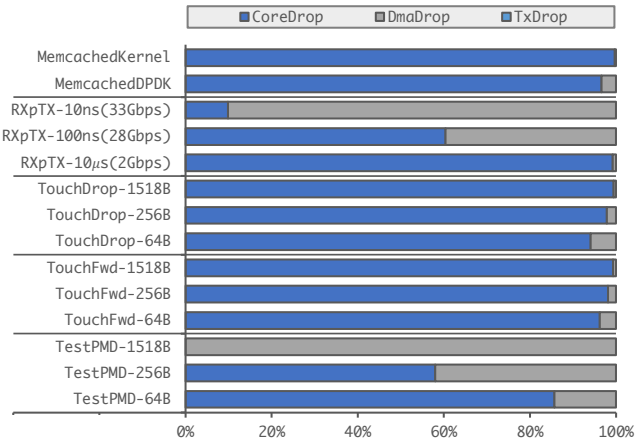


Fig. 5. The breakdown of packet drops when running `TestPMD`, `TouchFwd`, `RXpTX`, `MemcachedDPDK`, and `MemcachedKernel` at a high packet rate.

For `TestPMD`, when increasing the packet size, the drops shift from 85.7% CoreDrops with 64B packets to 100% DmaDrops with 1518B packets as the core is not the bottleneck for large packets. This shows that the DMA engine is the bottleneck when running `TestPMD` with large packets in gem5. For `TouchFwd`, increasing the packet size actually increases CPU utilization as the CPU should load the received packets from the memory into the register file before the packet can be forwarded. Therefore, most of the `TouchFwd` drops are CoreDrops. We run `TouchDrop` at the same rate at which we run `TouchFwd`. As shown in Fig.5, we see a

similar trend to TouchFwd with CoreDrops dominating at 93%, 97.9%, and 99.5% for 64B, 256B, and 1518B packets, respectively. Regarding RXpTX, we expect that increasing the processing time per packet on the CPU leads to more CoreDrops. This is confirmed in Fig.5; when we increase the processing time of RXpTX, the drops shift from DmaDrops to CoreDrops. The majority of drops in Memcached applications are CoreDrops.

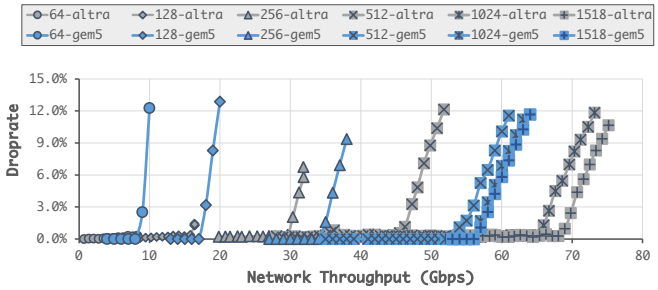


Fig. 6. Comparing the bandwidth vs. drop rate for `TestPMD` on gem5 and real system (`altra` configuration). The number in the legend show the packet size in bytes. The load generator software client cannot send enough load to `altra` with small packets.

B. Real-System vs. gem5

Figure 6 plots the bandwidth versus drop rate when running TestPMD with various packet sizes on both the real system (altra configuration) and gem5. We measure drop rates at the Pktgen application in altra and use EtherLoadGen in gem5. In gem5 simulations, the increase in packet drops can be solely attributed to TestPMD not being able to forward the packets quickly enough, as we employ a hardware load generator model. However, in real-system scenarios, drops can also occur due to the client load generator potentially acting as a bottleneck.

The first thing that stands out in Fig.6 is that the software load generator for altra becomes a bottleneck before TestPMD starts dropping packets. The load generator client is unable to load the DPDK server beyond 8Gbps and 16Gbps for 64B and 128B packets, respectively. gem5 also achieves slightly higher throughput and lower drop rate for packet sizes up to 512B. This clearly demonstrates the advantage of using a hardware load generator to properly load the server under test and eliminate any inaccuracies introduced into the measurements by the load generator application.

As shown in Fig.6, gem5 follows a very similar trend compared to altra for packet sizes up to 256B. Observing the graph, gem5's bandwidth saturates at around 53Gbps with 512B packet sizes, and increasing the packet size does not significantly reduce packet drops after reaching 56Gbps network bandwidth for 1518B packet sizes. This behavior is expected and suggests that TestPMD is core-bound for small packets. For packets larger than 512B, gem5 encounters a data movement bottleneck, either in the I/O bus (that loosely models a PCIe bus between the NIC and CPU) or in the memory subsystem. It is worth noting that by default, DCA is

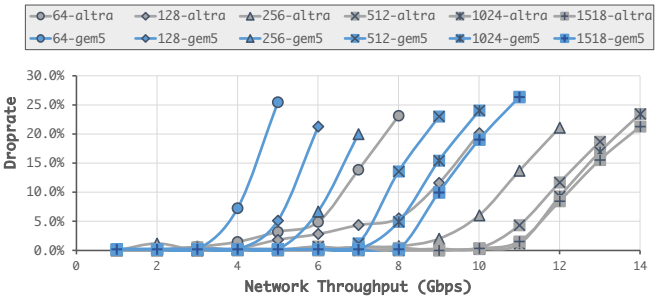


Fig. 7. Comparing the bandwidth vs. drop rate for `TouchFwd` on gem5 and real system (`altra` configuration). The number in the legend show the packet size in bytes.

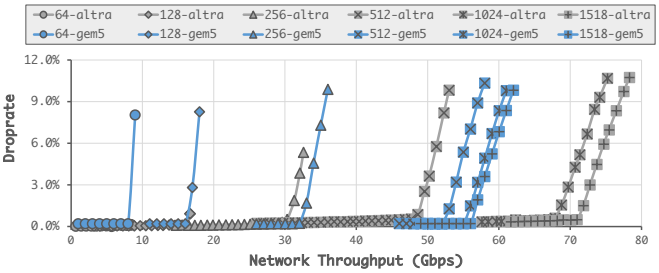


Fig. 8. Comparing the bandwidth vs. drop rate for `RXpTX` with processing time set to 10ns on gem5 and real system (`altra` configuration). The number in the legend shows the packet size in bytes.

enabled for gem5 (See Table I). If we disable DCA in gem5, the curve for gem5 in Fig.6 would shift to the left.

Figure 7 shows the bandwidth versus drop rate for TouchFwd. As expected, a server running TouchFwd experiences high drop rates at even lower bandwidth levels. One key takeaway from Fig.7 is the importance of minimizing data movement in the network software stack, which negatively affects network throughput. Additionally, we observe slightly lower throughput for gem5 across all packet sizes compared to altra. This difference is attributed to the core-bound nature of TouchFwd, which demonstrates the superior performance of a real Neoverse N1 core compared to its simulated counterpart in gem5. This observation underscores the impact of the underlying processor architecture on the performance of network-bound applications. Nevertheless, similar to TestPMD, we observe a strong correlation between altra and gem5 when running TouchFwd.

Figure 8 and Figure 9 illustrate the bandwidth versus drop rate for two RXpTX configurations, detailing the trends for both short CPU packet processing times (10ns) and long processing times (1μs), respectively. As expected, with 10ns processing time, RXpTX mirrors TestPMD's behavior in both altra and gem5 across all packet sizes. With 1μs processing time, RXpTX's performance aligns with that of TouchFwd for small packets. As depicted in Fig.9, the maximum sustainable bandwidth (MSB) drops to 2Gbps, 5Gbps, and 10Gbps for 64B, 128B, and 256B packets in gem5, whereas in altra, MSB drops to 3Gbps, 8Gbps, and 11Gbps for similar packet

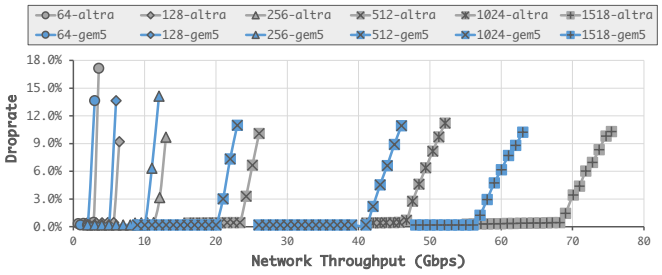


Fig. 9. Comparing the bandwidth vs. drop rate for `RXpTX` with processing time set to 1μs on gem5 and real system (`altra` configuration). The number in the legend show the packet size in bytes.

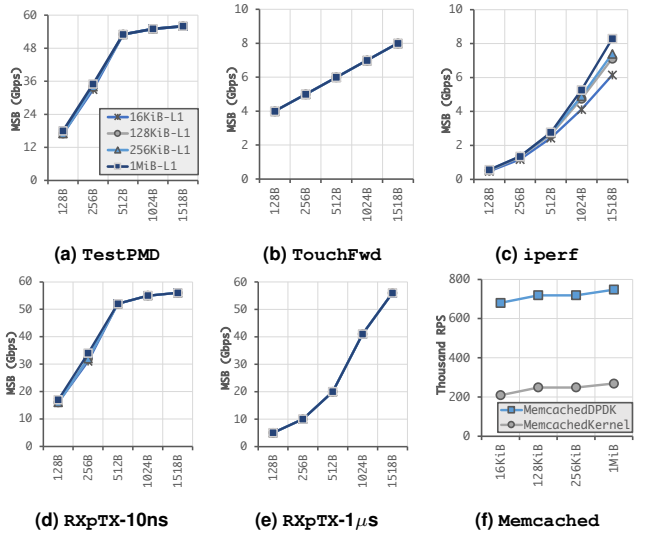


Fig. 10. Sensitivity of maximum sustainable bandwidth (MSB) and Request Per Second (RPS) to the L1 Cache sizes when running `TestPMD`, `TouchFwd`, `iperf`, `RXpTX` (with 10ns and 1μs processing times), `MemcachedDPDK`, and `MemcachedKernel` on gem5.

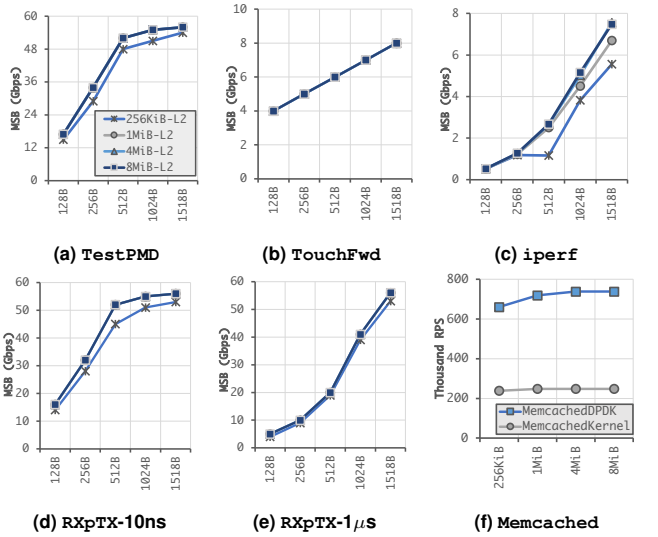


Fig. 11. Sensitivity of maximum sustainable bandwidth (MSB) and Request Per Second (RPS) to the L2 Cache sizes when running `TestPMD`, `TouchFwd`, `iperf`, `RXpTX` (with 10ns and 1μs processing times), `MemcachedDPDK`, and `MemcachedKernel` on gem5.

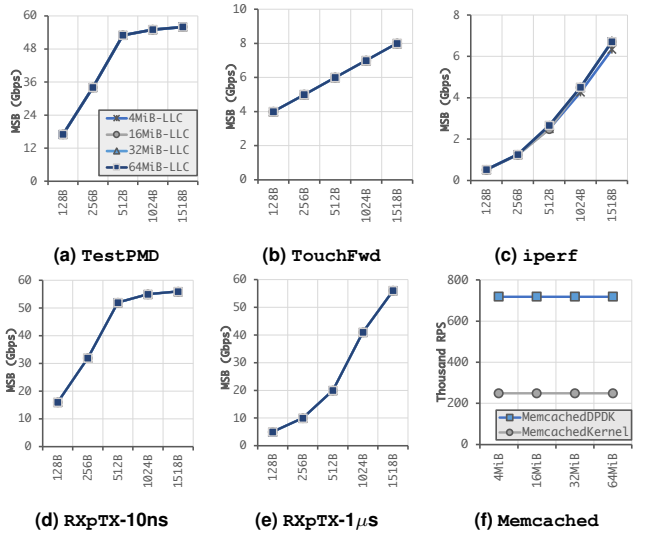


Fig. 12. Sensitivity of maximum sustainable bandwidth (MSB) and Request Per Second (RPS) to L3 Cache (LLC) sizes when running `TestPMD`, `TouchFwd`, `iperf`, `RXpTX` (with 10ns and 1μs processing times), `MemcachedDPDK`, and `MemcachedKernel` on gem5.

sizes. Larger packets do not suffer significant bandwidth degradation due to the higher CPU processing times, as their cost is amortized over many received packets. Once again, a strong correlation between gem5 and `altra` is observed.

C. gem5 Bandwidth Sensitivity

In this section, we analyze the sensitivity of network bandwidth to various microarchitectural configurations. Additionally, we evaluate the performance of the DPDK implementation in gem5 compared to the traditional Linux kernel network stack. To achieve this, we run `TestPMD`, `TouchFwd`, `iperf`, `RXpTX` (with 10ns and 1μs processing time), `MemcachedDPDK`, and `MemcachedKernel` on gem5 and plot the maximum sustainable bandwidth (MSB) or requests per second versus packet sizes while varying different microarchitectural configurations. We define MSB as the network bandwidth at the point on the bandwidth versus packet drop graph where the drop rate exceeds 1%. We use `iperf` as a representative application for comparing DPDK applications to an application that uses Linux kernel networking.

Sensitivity to Cache Size. Figure 10 shows the impact of the L1 cache sizes (both instruction and data) on the MSB. We observe that DPDK applications (`TestPMD`, `TouchFwd`, `RXpTX`-10ns, `RXpTX`-1μs) are not sensitive to L1 cache size across different packet sizes. However, `iperf`'s throughput is sensitive to L1 size for packets larger than 256B. `iperf` throughput with 1518B packets increases by 15.8% when increasing the L1 size from 16KiB to 128KiB. Because L2 is inclusive, we increase L1 size until 1MiB, which is the default

L2 size. Both MemcachedDPDK and MemcachedKernel show sensitivity to the L1 cache up to 1MiB.

As shown in Fig.11 decreasing L2 size to 256KiB results in a performance degradation for TestPMD and RXpTX-10ns. This suggests that DPDK working set size is larger than 256KiB and smaller than 1MiB. On the other hand, increasing L2 size from 1MiB to 4MiB still improves iperf performance, suggesting that Kernel stack working set size is larger than 1MiB. Also, we observe that MemcachedDPDK and MemcachedKernel throughput do not increase beyond 4MiB and 1MiB L2 sizes, respectively. This demonstrates the importance of the network stack on the sensitivity of microarchitectural configuration to performance.

In Fig.12, we do not observe any sensitivity between LLC size and the performance of the applications, even when sweeping LLC size up to 64MiB. This suggests low LLC contention when running a single network application. Larger LLCs can be beneficial for systems that co-run multiple LLC-intensive applications.

In summary, Fig.10, Fig.11 and Fig.12 illustrate that using a kernel-based network stack versus a userspace network stack exhibits different sensitivities to cache hierarchy configurations. This discrepancy arises due to variations in the working set sizes of DPDK and the Linux kernel software stack, coupled with diverse optimizations such as zero-copy networking, software prefetching, polling, the use of huge pages, and core pinning in the userspace network stack. These optimizations mask the anticipated performance gains associated with larger caches.

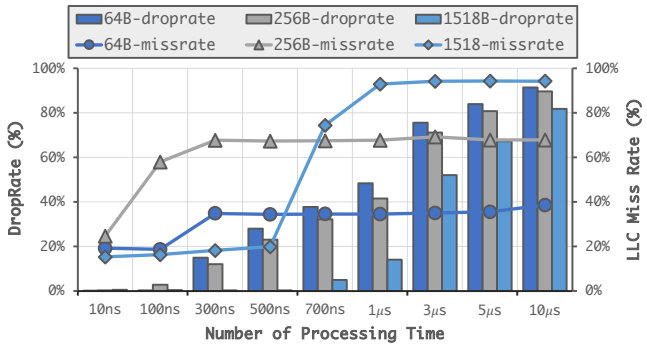


Fig. 13. Effect of the performance of various policies for DCA using 64B, 256B, and 1518B packets when running RXpTX with processing time set to 10ns, 100ns, 300ns, 500ns, 700ns, 1μs, 3μs, 5μs 10μs with ring buffer size set to 4096.

Sensitivity to DCA and Core Frequency. Figure 13 shows the effect of Direct Cache Access (DCA) using the RXpTX benchmark. We set the number of ring buffer entries to 4096 and fix the LLC size at 1MiB, while steadily increasing the number of CPU packet processing times in the application. The packet rate for each packet size is set to the value at their respective 10ns MSB. DCA uses 4 out of 16 ways of LLC for network data. This means that a maximum of 256KiB of LLC space is assigned to network data.

As we increase the number of processing time, beyond a

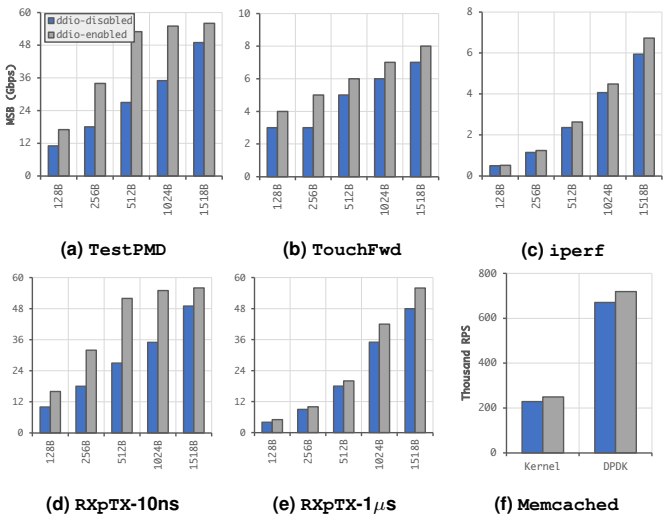
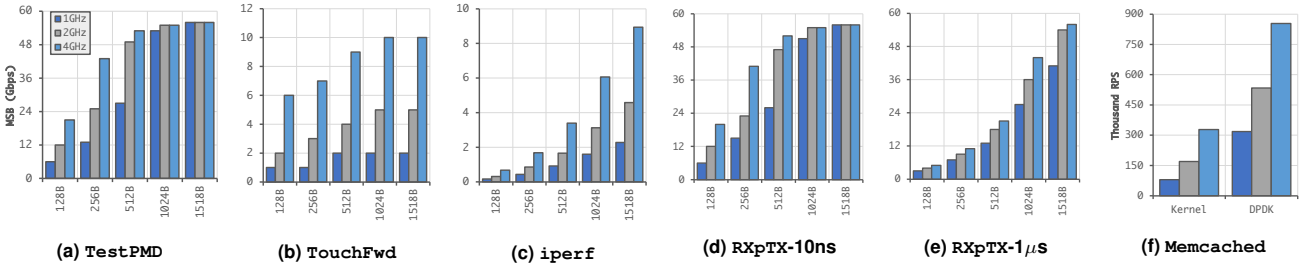


Fig. 14. Effect of Direct Cache Access (DCA) on Maximum Sustainable Bandwidth (MSB) when running TestPMD TouchFwd, iperf, RXpTX (with 10ns & 1μs processing time), MemcachedDPDK, & MemcachedKernel on gem5.

certain threshold, the core's packet processing rate begins to lag behind the RX rate from the network, and the RX Ring buffer starts to fill up until it becomes full, leading to packet drops. As shown in Fig.13, the processing time thresholds at which we start to experience packet drops for 64B, 256B, and 1518B packets are 300ns, 100ns, and 700ns, respectively. When the RX Ring begins to fill up, we also observe an increase in the LLC miss rate. This occurs because the 256KiB DCA space cannot accommodate the entire RX Ring buffer data, leading to DMA leaks [36].

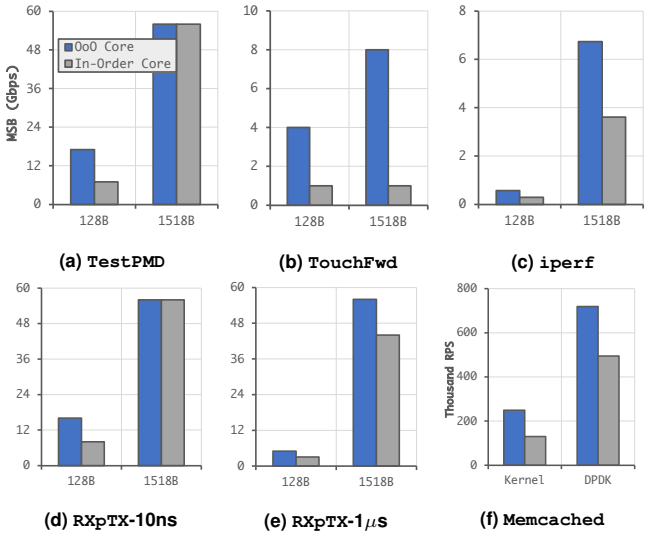
Figure 14 shows the sensitivity of network bandwidth to DCA. As expected, regardless of the application running or the packet sizes, DCA enables higher throughput due to the NIC's ability to DMA write/read packet data directly to/from the core's LLC. TestPMD achieves throughput increases of 54.5%, 88.9%, 96.3%, 57.1%, and 14.3% for 128B, 256B, 512B, 1024B, and 1518B packets, respectively, when DCA is enabled. TouchFwd sees throughput increases of 33.3%, 66.7%, 20.0%, 16.7%, and 14.3% for the same packet sizes. Similarly, RXpTX-10ns, RXpTX-1μs, and MemcachedDPDK experience throughput gains of up to 92.6%, 25%, and 7.2%, respectively. Finally, iperf (1518B packets) and MemcachedKernel observe a maximum throughput increase of 13.3% and 8.6%, respectively, when DCA is enabled. These results show that DCA provides higher relative performance improvement for DPDK applications compared to kernel-based networking applications. This is because DPDK implements zerocopy networking and minimizes memory copy operations which is the dominant end-to-end networking overhead in current kernel-based network stacks [37], [38].

Figure 15 depicts the sensitivity of network bandwidth to core frequency. MSB improves with higher core frequency when the application is core-bound. As the packet size for shallow network functions (TestPMD and RXpTX) increases, they transition from being core-bound to more IO-bound, and



(a) `TestPMD` (b) `TouchFwd` (c) `iperf` (d) `RXpTX-10ns` (e) `RXpTX-1`μ`s` (f) `Memcached`

Fig. 15. Effect of Core Frequency on Maximum Sustainable Bandwidth (MSB) when running `TestPMD` `TouchFwd`, `iperf`, `RXpTX` (with 10ns & 1μs processing time), `MemcachedDPDK`, & `MemcachedKernel` on gem5.



(a) `TestPMD` (b) `TouchFwd` (c) `iperf` (d) `RXpTX-10ns` (e) `RXpTX-1`μ`s` (f) `Memcached`

Fig. 16. Sensitivity of Maximum Sustainable Bandwidth (MSB) to CPU microarchitecture when running `TestPMD` `TouchFwd`, `iperf`, `RXpTX` (with 10ns & 1μs processing times), `MemcachedDPDK`, & `MemcachedKernel` on gem5.

thus the core's frequency has less impact on their MSB as their packet size increases. `TouchFwd`, being a deep network function, benefits from higher core frequency even at large packet sizes because the load on the core increases with the packet size. Interestingly, `iperf` exhibits similar behavior to a DPDK deep network function, and its performance, even for larger packets, improves with higher frequency. The MSB of both `MemcachedDPDK` and `MemcachedKernel` improves with higher frequency, which is expected as the memcached application is core-bound for the small dataset size that we run.

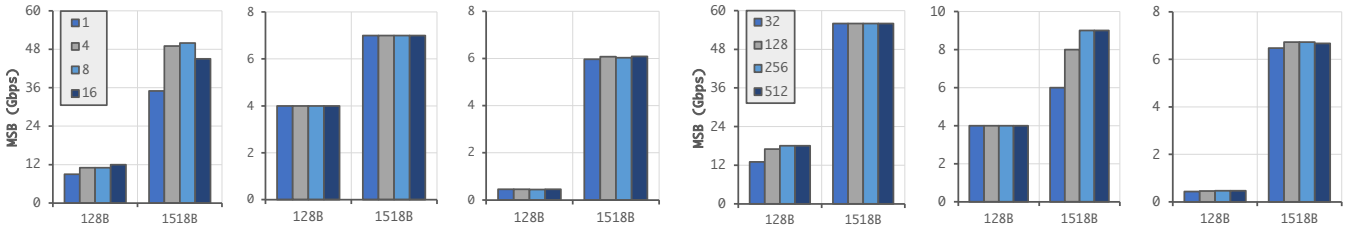
Sensitivity of Core Microarchitecture. Next, we explore the sensitivity of `TestPMD`, `TouchFwd`, `iperf`, `RXpTX` (with 10ns and 1us processing times), `MemcachedDPDK`, and `MemcachedKernel` to core microarchitecture parameters. Fig.16 shows the difference in MSB when comparing out-of-order and in-order cores . As demonstrated, since `TestPMD` and `RXpTX`-10ns at 1518B packet size are not core-bound, their MSB is insensitive to the core microarchitecture. We observe up to an 8×, 93.2%, 66.7%, 91.8%, and 45.3% increase in MSB or RPS for `TouchFwd`, `iperf`, `RXpTX`-10us, `MemcachedKernel`, and `MemcachedDPDK`, respectively, when using an out-of-order core.

Figure 17 shows the sensitivity of network applications to the number of memory channels (CH) and ROB entries in the out-of-order pipeline. We disable DCA for the memory channel sensitivity analysis (Fig.17a, Fig.17b, and Fig.17c) to ensure DRAM bandwidth utilization is apparent. As illustrated, increasing the number of memory channels for `TestPMD` with 1518B packets initially improves the MSB, but a degradation is observed after 8 channels. The decrease in MSB from 8 to 16 memory channels results from reduced row buffer locality. `MemcachedKernel` sees an improvement of 8.6% moving from 1 to 4 channels, with `MemcachedDPDK` showing no sensitivity to the number of channels. `TouchFwd`'s MSB with 1518B packets increases by 33.3% when increasing the ROB size from 32 to 128. Similarly, `RXpTX`-10ns demonstrates a maximum MSB gain of 30.8% for 128B packets when moving from 32 to 256 ROB entries. These improvements are due to having a long chain of dependent load and store in the pipeline and improvements in memory-level parallelism with larger ROB sizes. We found that ROB sizes of 32 and 128 are sufficient to sustain the performance of `MemcachedDPDK` and `MemcachedKernel`, respectively.

Benchmarking with Real Applications. Figure 18 plots the throughput vs. packet drop rate for `MemcachedDPDK` and `MemcachedKernel`. As shown, `MemcachedDPDK` and `MemcachedKernel` achieve 709kRPS and 218kRPS before their drop rate shoot up, respectively.

Figure 19 illustrates the sensitivity of Memcached response latency to core clock frequency. `EtherLoadGen` timestamps a request packet and measures round-trip latency without affecting application's performance. The latency values presented in Fig.19 are normalized to a core clocked at 3GHz. As shown, we see that for both `MemcachedKernel` and `MemcachedDPDK` the response time at high rates significantly increases when reducing the frequency of the core. Note that as soon as packet drops begin, the latency numbers reported by `EtherLoadGen` often decrease because the dropped packets no longer contribute to the latency sampling.

Simulation Speedup with `EtherLoadGen`. Here, we evaluate the performance benefit of using our Hardware `EtherLoadGen` model to replay packet traces for benchmarks explained in Sec.V compared with using gem5 in



(a) `TestPMD` [CH] (b) `TouchFwd` [CH] (c) `iperf` [CH] (d) `TestPMD` [ROB] (e) `TouchFwd` [ROB] (f) `iperf` [ROB]

Fig. 17. Sensitivity of packet Maximum Sustainable Bandwdith (MSB) to the number of DRAM memory channels (CH), and number of reorder buffer entries (ROB) in the OoO CPU pipeline when running `TestPMD`, `TouchFwd`, and `iperf` on gem5.

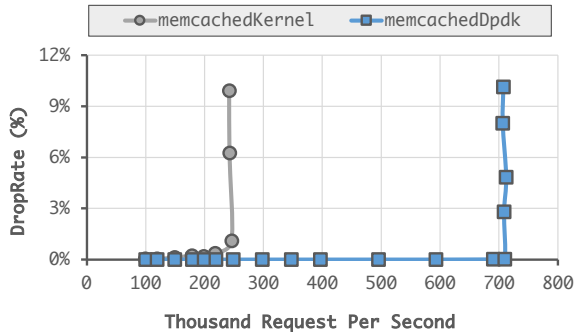
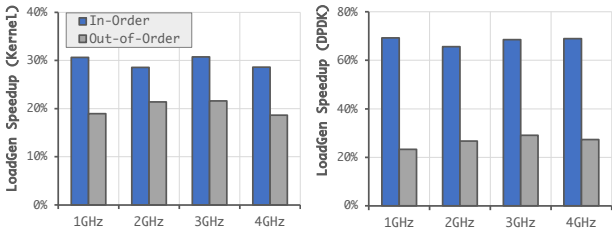
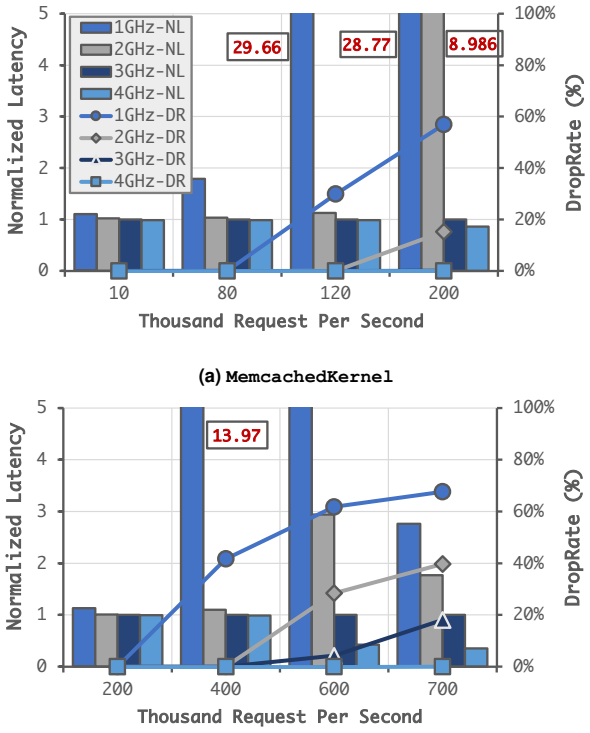


Fig. 18. Comparing throughput vs drop rate of `MemcachedKernel` and `MemcachedDPDK` running in gem5.



(a) kernel speedup (b) DPDK speedup

Fig. 20. Simulation Time speedup when using `EtherLoadGen` compared to dual mode running `MemcachedKernel` and `MemcachedDPDK`.



(a) `MemcachedKernel`

(b) `MemcachedDPDK`

Fig. 19. Sensitivity of `MemcachedKernel` and `MemcachedDPDK` response time and drop rate to core's clock frequency.

dual mode and run a software load generator. Here, we run memcached using a functional CPU during the warmup phase and more detailed in-order and out-of-order core in the request phase. Fig.20 shows that `EtherLoadGen` achieves up to a 70% speedup in simulation over dual mode gem5.

VIII. Conclusion

Identifying the current gap in gem5's capability to simulate state-of-the-art userspace network software stacks, we enhanced gem5 to simulate DPDK-based network applications. Additionally, we integrated a hardware load generator model to simplify and accelerate network simulation. We further developed a suite of networking microbenchmarks to rigorously test the networking stack. Utilizing our extensions, we thoroughly characterized packet processing in the DPDK userspace networking stack, achieving speeds exceeding 50 Gbps per core. We compared these results with those from a real ARM server. Lastly, we contrasted the sensitivities of Linux kernel networking and DPDK to various microarchitectural settings, highlighting the importance of userspace network modeling in gem5. Our extensions are open-source and available at https://github.com/architecture-research-group/gem5-dpdk-setup.

Acknowledgments

This work was supported in part by grants from National Science Foundation (CCF-2239020, OAC-2311891), and ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We thank NVIDIA Academic Hardware Grant Program and Ampere Computing for their hardware donations.

References

[1] M. Wade, E. Anderson, S. Ardalan, *et al.*, "TeraPHY: A chiplet technology for low-power, high-bandwidth in-package optical I/O," *IEEE Micro*, vol. 40, no. 2, pp. 63–71, 2020.

[2] Q. Cai, M. Vuppalapati, J. Hwang, C. Kozyrakis, and R. Agarwal, "Towards s tail latency and terabit ethernet: Disaggregating the host network stack," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM '22, Amsterdam, Netherlands: Association for Computing Machinery, 2022, pp. 767–779, ISBN: 9781450394208. DOI: 10.1145/3544216.3544230. [Online]. Available: https://doi.org/10.1145/3544216.3544230.

[3] J. Cabal, P. Benáček, L. Kekely, M. Kekely, V. Puš, and J. Kořenek, "Configurable fpga packet parser for terabit networks with guaranteed wire-speed throughput," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, Monterey, CALIFORNIA, USA: Association for Computing Machinery, 2018, pp. 249–258, ISBN: 9781450356145. DOI: 10.1145/3174243.3174250. [Online]. Available: https://doi.org/10.1145/3174243.3174250.

[4] E. Y. Jeong, S. Woo, M. Jamshed, *et al.*, "Mtcp: A highly scalable user-level tcp stack for multicore systems," ser. NSDI'14, Seattle, WA: USENIX Association, 2014, pp. 489–502, ISBN: 9781931971096.

[5] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, Shanghai, China: Association for Computing Machinery, 2017, pp. 325–341, ISBN: 9781450350853. DOI: 10.1145/3132747.3132780. [Online]. Available: https://doi.org/10.1145/3132747.3132780.

[6] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14, Broomfield, CO: USENIX Association, 2014, pp. 49–65, ISBN: 9781931971164.

[7] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, pp. 361–378, ISBN: 978-1-931971-49-2. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[8] M. Marty, M. de Kruijf, J. Adriaens, *et al.*, "Snap: A microkernel approach to host networking," in *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.

[9] L. Rizzo and M. Landi, "Netmap: Memory mapped access to network devices," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 422–423, Aug. 2011, ISSN: 0146-4833. DOI: 10.1145/2043164.2018500. [Online]. Available: https://doi.org/10.1145/2043164.2018500.

[10] Intel®, *DPDK Intel NIC Performance Report Release 17.08*, http://fast.dpdk.org/doc/perf/DPDK_17_08_Intel_NIC_performance_report.pdf.

[11] N. Binkert, B. Beckmann, G. Black, *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[12] J. Lowe-Power, A. M. Ahmad, A. Akram, *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[13] S. Karandikar, H. Mao, D. Kim, *et al.*, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42. DOI: 10.1109/ISCA.2018.00014.

[14] S. Agarwal, R. Agarwal, B. Montazeri, *et al.*, "Understanding host interconnect congestion," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, ser. HotNets '22, Austin, Texas: Association for Computing Machinery, 2022, pp. 198–204, ISBN: 9781450398992. DOI: 10.1145/3563766.3564110. [Online]. Available: https://doi.org/10.1145/3563766.3564110.

[15] *TCPDUMP LIBPCAP*, https://www.tcpdump.org/.

[16] *dpdk-pdump Application*, https://doc.dpdk.org/guides/tools/pdump.html.

[17] L. Tianhua, Z. Hongfeng, C. Guiran, and Z. Chuansheng, "The design and implementation of zero-copy for linux," in *2008 Eighth International Conference on Intelligent Systems Design and Applications*, vol. 1, 2008, pp. 121–126. DOI: 10.1109/ISDA.2008.102.

[18] *Intel 8254x*, https://wiki.osdev.org/Intel_8254x.

[19] A. Mohammad, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim, "Dist-gem5: Distributed simulation of computer clusters," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2017, pp. 153–162.

[20] S. AION, *Driving innovation and accelerating next-gen technology deployments with award-winning L2-7 Network and Cloud testing solutions.* https://www.spirent.com/products/aion.

[21] M. Primorac, E. Bugnion, and K. Argyraki, "How to measure the killer microsecond," in *Proceedings of the Workshop on Kernel-Bypass Networks*, ser. KBNets '17, Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 37–42, ISBN: 9781450350532. DOI: 10.1145/3098583.3098590. [Online]. Available: https://doi.org/10.1145/3098583.3098590.

[22] K. Technologies, *High-Volume Traffic Generator Products Catalog*, https://www.keysight.com/us/en/assets/

7121-1065/catalogs/High-Volume-Traffic-Generator-Products-Catalog.pdf.

[23] M. Alian, Y. Yuan, J. Zhang, R. Wang, M. Jung, and N. S. Kim, "Data direct I/O characterization for future I/O system exploration," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2020, pp. 160–169.

[24] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: A cautionary tale," in *3rd Symposium on Networked Systems Design & Implementation (NSDI 06)*, San Jose, CA: USENIX Association, May 2006. [Online]. Available: https://www.usenix.org/conference/nsdi-06/open-versus-closed-cautionary-tale.

[25] *pcap(3) - Linux man page*, https://linux.die.net/man/3/pcap.

[26] *Packet Capture Library*, https://doc.dpdk.org/guides/prog_guide/pdump_lib.html.

[27] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, IEEE, 2009, pp. 33–42.

[28] *Testpmd Application User Guide*, https://doc.dpdk.org/guides/testpmd_app_ug/.

[29] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, "Deep packet inspection as a service," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14, Sydney, Australia: Association for Computing Machinery, 2014, pp. 271–282, ISBN: 9781450332798. DOI: 10.1145/2674005.2674984. [Online]. Available: https://doi.org/10.1145/2674005.2674984.

[30] M. Alian, S. Agarwal, J. Shin, *et al.*, "IDIO: Network-driven, inbound network data orchestration on server processors," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 480–493. DOI: 10.1109/MICRO56248.2022.00042.

[31] *memcached - a distributed memory object caching system*, https://memcached.org.

[32] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct cache access for high bandwidth network i/o," in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005, pp. 50–59. DOI: 10.1109/ISCA.2005.23.

[33] AmpereComputing®, *DPDK setup and tuning guide for Ampere Altra Processors*, https://amperecomputing.com/tuning-guides/DPDK-setup-and-tuning-guide.

[34] Intel®, *The Pktgen Application*, https://pktgen-dpdk.readthedocs.io/en/latest/commands.html#runtime-options-and-commands.

[35] *Buildroot: Making embedded Linux easy*, https://buildroot.org/.

[36] M. Alian, S. Agarwal, J. Shin, *et al.*, "Idio: Network-driven, inbound network data orchestration on server processors," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 480–493. DOI: 10.1109/MICRO56248.2022.00042.

[37] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 65–77, ISBN: 9781450383837. DOI: 10.1145/3452296.3472888. [Online]. Available: https://doi.org/10.1145/3452296.3472888.

[38] R. Chen and G. Sun, "A survey of kernel-bypass techniques in network stack," in *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, ser. CSAI '18, Shenzhen, China: Association for Computing Machinery, 2018, pp. 474–477, ISBN: 9781450366069. DOI: 10.1145/3297156.3297242. [Online]. Available: https://doi.org/10.1145/3297156.3297242.