Fine-Grained Isolation of the TCP/IP Stack with Language Safety

Undergraduate Thesis

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Tirth Jain
ID No. 2019A7TS0120P

Under the supervision of:

Dr. Anton Burtsev
&
Dr. K Hari Babu

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS
December 2022

Declaration of Authorship

I, Tirth Jain, declare that this Undergraduate Thesis titled, 'Fine-Grained Isolation of the TCP/IP Stack with Language Safety' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Certificate

This is to certify that the thesis entitled, "*Fine-Grained Isolation of the TCP/IP Stack with Language Safety*" and submitted by Tirth Jain ID No. 2019A7TS0120P in partial fulfillment of the requirements of BITS F421T Thesis embodies the work done by him under my supervision.

Supervisor	*Co-Supervisor*
Dr. Anton Burtsev	Dr. K Hari Babu
Asst. Professor,	Asst. Professor,
University of Utah	BITS Pilani Pilani Campus
Date:	Date:

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS

Abstract

Bachelor of Engineering (Hons.)

Fine-Grained Isolation of the TCP/IP Stack with Language Safety

by Tirth Jain

Even after decades of work to make monolithic kernels more secure, serious vulnerabilities in them are still reported every year. Because the entire monolithic kernel is in one address space, an attacker is just one vulnerability away from owning the entire machine. We argue that it is time to decompose monolithic kernels like Linux into smaller parts that run in isolated compartments and communicate using secure interfaces. We think this is timely due to growing hardware and software support of isolation.

In this Thesis, I discuss Atmosphere, our approach to microkernelization and fault isolation of an operating system kernel. Specifically, my work focuses on the network stack which is part of our larger effort to build a new operating system, Atmosphere. We argue that the network stack is a source for bugs and that isolation is the way forward to minimize the impact of these bugs.

Contents

Chapter 1

Introduction

Operating system reliability is a topic that has been studied for decades but nevertheless remains a major concern today. Even thought its been 30 years since Linux kernel was developed, critical vulnerabilities are being found in the kernel. On top of that, the monolithic design (i.e., everything in the kernel is running in a single address space) makes it such that the attacker is only one exploit away from taking control of the entire system. This is concerning because device drivers, often developed by third parties are a major source of vulnerabilities. Redundancy in hardware and software protection mechanisms can protect against transient faults but persistent faults because of undetected logic flaws still remain a major concern. In recent times, with the advent of cloud services, this becomes even more significant because clouds often have to run untrusted code that needs to be isolated from other parts of the operating system.

Logic errors and faults can propogate from within a component and propogate to other parts of the system. Microkernelization aims to solve this problem by isolating various parts of an operating system and minimizing the effect of failures in a single subsystem. In a microkernel, only the bare minimum that is required to boot an operating system is included in the kernel. All other components such as the filesystem, the device drivers, and the network stack are run in separate isolated processes that interact with each other using some form of IPC. Inter component fault propogation can thus be reduced by introducing safe IPC mechanisms.

Atmosphere is a microkernel based operating system. In this thesis, I discuss the design of Atmosphere mainly focusing on the the network stack and how its design can be used to model other services of the operating system. The network stack was built with two main principles in mind: maximizing fault isolation and minimizing its cost on performance. For the first principle, we take microkernelization to its extremes and build a per connection (or per socket for UDP) network stack. This means, every new connection has its own TCP/IP stack that can process and dispatch packets. This ensures that if one connection is corrupted, the rest of the connections on the system can keep operating. Although this is not completely possible since there is some

inherent shared state on any host machine on a network. In later sections we discuss how we minimize this shared state. For the second principle, we implement a zero copy model and explain how using Rust's guarantees help us implement this with confidence. On failure, a component can be safely restarted and meanwhile, all IPC calls to it can return an error.

The rest of this Thesis is organised as follows:

- In Background and Related Work we take a look at the contemporary approaches to microkernelization.
- In Isolation Mechanisms we take a detour to discuss modern software and hardware isolation mechanisms and why we choose Rust.
- The Network Stack discusses the design principles of Atmosphere.
- Evaluation talks about the API design and the internals of the network stack we built.
- **??** evaluates the performance of the network stack as compared to the current standards.

Chapter 2

Background and Related Work

Tirth *some intro here? idk*

2.1 Theseus

Theseus[2] is a microkernel based operating system written in Rust. Rust's safety guarantees enable Theseus to run all software written in Rust, including userspace applications, to run in a single address space and at a single privilege level. Thus, eliminating the need for virtual memory management and protection rings. Theseus introduces the idea of cells that are described as a software-defined unit of modularity that serves as the core building block of the OS. A cell can be modeled as a crate in Rust. On booting, only the microkernel, ie, the `nano_core` is loaded which bootstraps the system. All other cells are dynamically loaded on demand. Theseus piggybacks on Rust's safety guarantees to enable reliable IPC between cells. For example, memory mappings are a 1-1 mapping to a physical frame. They can only be shared behind a read-only `&MappedPages` reference eliminating the double-free and the use-after-free problem.

The most important contribution by Theseus is the idea of eliminating (or minimizing) shared state between components. All OS services can be modelled as servers and applications requesting these services are modelled as clients. Theseus eliminates state-spill[1] across cells by eliminating all state from the servers. Everything that is needed to service a client's request is stored in the client itself and thus the servers can be completely stateless. This allows a server cell to be replaced by a new cell without any state loss. This, however, is not always possible. In case a state cannot be eliminated, as in the case of descriptor tables, the state is stored in a `state_db`. The `state_db` is a key-value database that stores states with a static lifetime that the server can request a (weak) reference to. In case a server needs to be restarted, its state can be recovered from the `statedb`. The statelessness of cells also allows for live updates. A server can be replaced,

ie, a patch can be applied to a component without having to restart the entire operating system. Applying a patch is as simple as swapping a cell with a new cell.

Tirth *Maybe add limitations and performance evaluations of Theseus here?*

2.2 Singularity OS

Singularity[**singularity**] introduced "Software Isolated Processes" (SIPs) which use software verification instead of hardware protection mechanisms to isolate processes. SIPs cannot have shared memory. Instead, data can be passed between SIPs using an "exchange heap". Data on the exchange heap can be owned only by a single process but the ownership can be "transferred". Static verification ensure that programs do not try to access an object after it has been passed (ie a dangling pointer). Ownership can be transferred between SIPs using "Contract-Based Channels". Channels are described using statically defined interfaces in the Sing# language. The communicating SIPs act as state machines with clearly defined states and the messages that can be passed on each state. Once a message has been passed, its data can no longer be used by the sending SIP. Ownership of data on the exchange heap is recorded so that blocks can be freed on process termination preventing memory leaks. This process isolation allows singularity to run the kernel and all SIPs in a single physical address space.

All programs running on Singularity must ship with a manifest. Manifest-based programs clearly define their resource requirements, desired capabilities and dependencies on other programs. The manifest can be used by the system to ensure that the program's requirements can be met and that the program satisfies all correct usage guarantees. The absence of shared memory and static verification of all communication makes the creation and termination of SIPs inexpensive.

2.3 RedLeaf

Chapter 3

Isolation Mechanisms

Tirth *Some intro here*

3.1 Memory Protection Keys

Recently, Intel introduced Memory Protection Keys for isolation support in the hardware. The 4 unused bits in the page table are used to assign a protection key to a page. The PKRU register contains a 32-bit key representing the read/write permissions for 16 possible keys. The PKRU register can be accessed using user-mode `WRPKRU` (for writing) and `RDPKRU` instructions. For process sandboxing, the trusted computing base (TCB) can assign permissions to memory by writing the protection key to the PKRU register before passing it to the untrusted domain. To ensure an untrusted domain cannot use these instructions, we can use binary rewriting to remove all occurences of `WRPKRU` and `RDPKRU`.

Process isolation with MPK incurs a very low overhead. Crossing domains takes domains takes only 20-26 cycles [**ipc-62**, **ipc-33**] and passing a buffer to an untrusted domain is simply manipulating bits in a global table that holds PKRU values for every domain. However, the number of domains that can co-exist is limited by the number of possible PKeys supported by the hardware.

3.2 Native Client

Introduced in 2009 by Google (now deprecated in favour of WebAssembly), Native Client (NaCl) added support for sandboxing untrusted native x86 code in browsers. NaCl limits the address space of a domain to a 4GiB segment. The two main invariants enforced by NaCl are: no loads or stores can access data outside their 4GiB segment, and all jumps need to land to a valid

instruction boundary inside the domain. The R15 register is reserved as RZP which always points to the start of the domain. NaCl introduces pseudo-instructions that are finally expanded into x86 instructions that maintain the above invariants. All addresses are modified this way: the first 32 bits of the register are masked and then the register is added with R15. This way, every address lies in its respective segment. In addition to this, for jump instructions, the last 5 bits of the address register are masked so that the jump destination is always 32 bytes aligned. This is done to make sure that the masking instructions are not bypassed by jumping to an invalid target.

The NaCl runtime incurs an overhead of only 10-15%. Since, Google NaCl only supports a single untrusted domain, we adapt Google NaCl to support multiple domains. Data can be passed between domains by copying the buffer between segments.

To evaluate how NaCl fares in a multi-domain setup, we implement a version that supports multiple domains. To compare the performance on ARM CPUs, we implement a similar setup in our evaluations.

3.3 LXFI

LXFI is a capability-based Software Fault Isolation mechanism that allows multiple untrusted domains to co-exist. A capability can be of three kinds: READ, WRITE and CALL denoting read, write and execute permissions respectively. Capabilities are stored in a hash-table which binds a memory region with the domain that owns it. LXFI uses compile-time rewriting to enforce a capability check before every load/store operation and every call or indirect jump to ensure that the domain has the capability to access the data or call the function.

3.4 ARM Pointer Authentication

ARM introduced Pointer Authentication alongwith Memory Tagging Extensions (described in the next section) in ARMv8.3. PAC adds optional support for cryptographically signing pointers with a context. This context comprises of 5 keys: 2 for data, 2 for code and one user defined key. ARM provides the PAC* and AUT* instructions for signing and authenticating a signed pointer respectively. PACMem [**PACMem**] describes how PAC can be used to achieve memory safety. PACMem maintains a metadata table which stores the base address, the size and a random "birthmark" of every object. When a pointer is dereferenced, it validates the metadata agains the table. Once the object is deallocated, the corresponding metadata table entry is deleted. This makes sure that the object cannot be used later since the authentication will fail.

Passing data has a low overhead as it is as simple as passing the PAC alongwith the pointer. However, the runtime incurs a high overhead since every dereference involves an expensive hash-table lookup.

3.5 ARM Memory Tagging Extensions

Similar to MPK, MTE allows memory regions to be tagged with one of 16 possible tags. However, MTE tags live in an inaccessible region of physical memory. MTE also allows for more fine-grained isolation as it works on 16 byte granules as compared to the page sized granularity in MPK. In our design, every domain can be assigned a tag. We use the 'bfi' instruction to tag every pointer with it's corresponding tag. To pass buffers between domains, the buffer has to be retagged with the tag of the new domain.

The runtime overhead is low since the `bfi` instruction takes very few cycles to tag the pointer. The fine-grained granularity offered by MTE comes at a cost because retagging large buffers in memory is essentially tagging 16 (64 in the kernel) bytes in a loop.

3.6 Evaluating Isolation Mechanisms

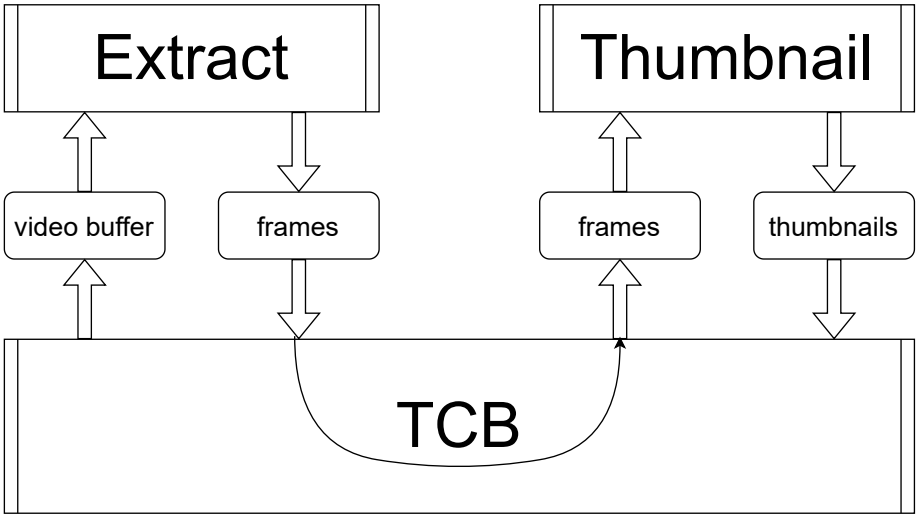

FIGURE 3.1: Video Processing Pipeline

To evaluate the aforementioned isolation mechanisms, we implement a video processing pipeline with two steps: (1) Frame Extractor which decodes frames from a video buffer and (2) Thumbnail encoder which converts the extracted frames into a suitable pixel format and encodes them into a GIF. We implement this pipeline using FFmpeg libraries in C and apply 5 different isolation

mechanisms: SSFI, NaCl, MPK, MTE, and PAC. ?? describes the experiment setup and ?? compares the performance of the setups as compared to an unisolated setup.

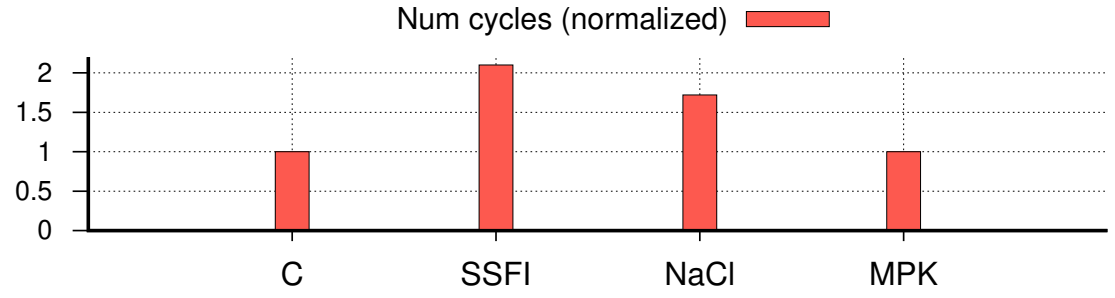

FIGURE 3.2: Overhead of FFmpeg (x86)

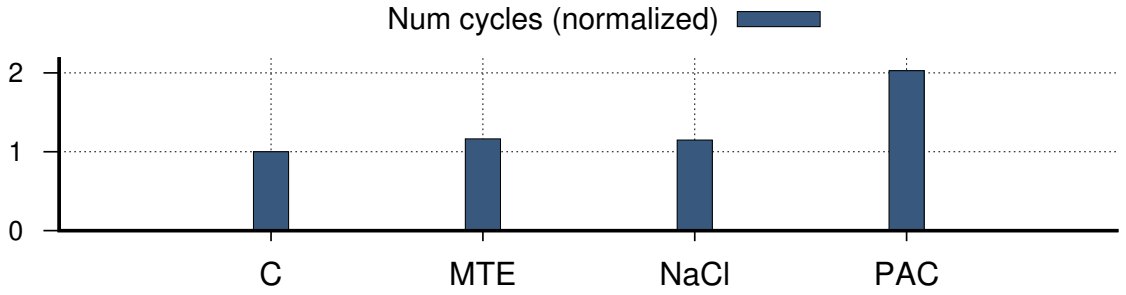

FIGURE 3.3: Overhead of FFmpeg (ARM)

With Intel MPK and ARM MTE, the extracted frames are retagged and passed between the compartments. However, with NaCl, they are copied to the second compartment, resulting in an extra overhead. MPK comes close to the performance of non-isolated C. SSFI schemes suffer from overheads of memory copying (NaCl on x86) and expensive retagging (MTE on ARM).

Tirth *add a sentence about why NaCl performs better on ARM than on x86*

We also use PAPI to further breakdown the overheads of each isolation mechanism on x86 CPUs.

(in Millions)	C	SSFI	NaCl	MPK
Cycles	5995	11578	10188	5996
Instructions	14681	34797	25846	14681
Branches	905	938	907	905
Branches mispred.	60	63	56	60
loads	3267	3844	3448	3267
stores	1415	1594	1493	1415

TABLE 3.1: Microarchitectural comparison of SFI and C on FFmpeg-x86 benchmark.

Chapter 4

The Network Stack

In this chapter, we take a look at how we implemented the UDP stack for Atmosphere. Since Atmosphere is not ready yet, we build a standalone network stack that runs in the userspace and evaluate its performance. We use a custom userspace network driver (based on Ixy) to process raw ethernet frames sent to and from the network stack. This driver is shared across applications just as it would be in the real Atmosphere.

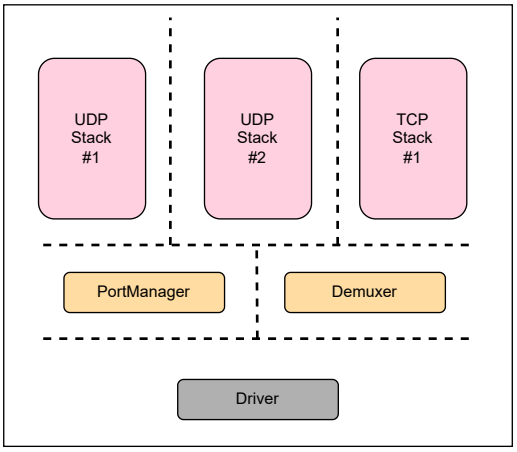

FIGURE 4.1: Atmosphere network stack design

4.1 Driver for the Network Card

Ixy is a network driver implemented in the userspace for Intel's Ixgbe family of network cards that shows how network cards work at the driver level. It is implemented in a fashion similar to DPDK and Snabb. We use a version of Ixy written in Rust and modify it to fit the design of our network stack. Two main limitations of Ixy (as per our design) are: (1) it handles allocations in

the driver and (2) received packets cannot be reused as transmit packets. (2) is important as for a lot of applications (e.g. MICA), we want to be able to flip headers in a received packet and send it back to the client.

The major modifications we make to ixy.rs are:

- Change the way packets are allocated. We get rid of the huge-pages backed Mempool and allow packets to be discretely allocated. DMA mapping is done for every packet.
- Receive queues do not allocate packets themselves. Instead, the application supplies a batch of allocated buffers which is used to read packets from the NIC. This relieves the NIC from having to do any allocations.

On testing with an Ixgbe NIC with a line rate of 10GB/s, we see no drop in performance after our modifications.

4.1.1 Packet Storage

As mentioned above, packets are stored in a DMA-mapped page-sized buffer. Listing 1 illustrates the Packet struct and how it is allocated. While allocating a page for each packet is wasteful in terms of memory, there is no discernible change in performance in processing packets. This also enables the NIC to interleave packets from different applications in the same queue without crossing any isolation boundaries.

4.2 Port Manager

The PortManager handles all allocations and deallocations of ports and keeps track of which stacks they are bound to. To be able to formally verify its implementation in the future, the interface it exposes is minimal and it only interacts with the opaque identifiers rather than actual types. UDP stacks can be identified by using only the destination port in an incoming packet (we assume that the device is connected to a single IPv4 interface for simplicity). But for TCP connections, since a single port can handle multiple connections, and every connection is handled by a different stack, we maintain a separate flow table for matching TCP connections. The PortManager also helps the demuxer identify which stack a packet should go to, or if it is to be dropped.

```rust
#[repr(C, align(PAGESZ))]
pub(crate) struct PacketBuffer {
    data: [u8; PAGESZ]
}

pub struct Packet {
    pub(crate) addr_virt: *mut u8,
    pub(crate) addr_phys: usize,
    pub(crate) len: usize,
    pub(crate) data: Box<PacketBuffer>,
}

pub fn alloc_pkt(len: usize) -> Option<Packet> {
    let mut buffer = Box::new(PacketBuffer{ data: [0;PAGESZ] });
    let addr_virt = buffer.data.as_mut_ptr();
    let mut p = Packet {
        data: buffer,
        addr_virt,
        len,
        addr_phys: 0,
    };
    match vfio_map_dma(p.addr_virt as usize, PAGESZ) {
        Ok(addr_phys) => {
            p.addr_phys = addr_phys;
            Some(p)
        }
        Err(e) => {
            error!("{}", e);

            None
        }
    }
}
```

Listing 1: Packet storage in modified ixy

4.3 Demuxer

The demuxer matches incoming packets with a destination stack and and returns free buffers from the NIC to the stack that owns them. The demuxer can also maintains private queues for applications that are not actively receiving packets but have incoming packets destined to them. On receiving a packet, the demuxer parses the protocol and the destination port (or the TCP 5-tuple) and use the PortManger to find the destination stack. If the packet is matched, we push it to the private queue. The next time the application calls receive, the private queue is emptied and the packets are returned to the stack. If the private queue for a stack is saturated, incoming packets are simply dropped.

4.3.1 Packet Flipping

Since the buffers for a receive call come from a stack, and since all packets received might not be destined to the same stack, the demuxer can flip a buffer from its private queue so that the application does not lose a free buffer to the demuxer. This is done by popping a buffer from a private queue and flipping it with the received buffer. The application can then get a free buffer in return for the buffer it contributed for the packet of another stack.

4.3.2 UDP Interface

We provide a low level interface for the application to interface applications with the udp protocol. We provide a `UdpStack` interface that exposes two main methods: `tx_batch()` and `rx_batch()` both of which are non-blocking and interact with discrete packet buffers. Note that this is currently a low level interface requiring the user to setup a static arp table and routes in the ipv4 table themeselves. this can be changed in the future when the stack is integrated with atmosphere. We also provide a convenience function called `prepare_batch()` that lets users provide a buffer and fills discrete packets with all required headers and the data from the given buffer. For ease of manipulating packet headers, we provide a `UDPPacketRepr` representation which has setters and getters for protocol headers. 2 illustrates the code for a simple UDP echo server.

```rust
pub fn main() {
    let mut stack = create_udp_stack(Ipv4Address::new(10, 0, 0, 2), 5000);
    let mut free_bufs: VecDeque<RawPacket> = VecDeque::with_capacity(NUM_PACKETS);
    let mut recv_batch: VecDeque<UdpPacketRepr> = VecDeque::with_capacity(NUM_PACKETS);
    let mut send_batch: VecDeque<UdpPacketRepr> = VecDeque::with_capacity(NUM_PACKETS);

    let mut num_recvd;

    loop {
        (num_recvd, recv_batch, free_bufs) = stack
            .recv_batch(recv_batch, free_bufs, BATCH_SIZE)
            .expect("failed to receive packets");
        recv_batch.drain(..num_recvd).for_each(|mut pkt| {
            // flip source and destination
            pkt.set_udp_packet(|mut udp| {
                let src = udp.get_source();
                let dst = udp.get_destination();
                udp.set_destination(src);
                udp.set_source(dst);
            });
            pkt.set_ip_packet(|mut ip| {
                let src = ip.get_source();
                let dst = ip.get_destination();
                ip.set_destination(src);
                ip.set_source(dst);
            });
            send_batch.push_back(pkt);
        });
        (send_batch, free_bufs) = stack
            .send_batch(send_batch, free_bufs)
            .expect("failed to sent packets");
    }
}
```

Listing 2: Example UDP echo server

Chapter 5

Evaluation

To evaluate the performance of our network stack, we run experiments on the Cloudlab network testbed. We utilize two c220g2 servers configured with two Intel E5-2660 v3 10-core Haswell CPUs, 160 GB RAM, and a dual-port Intel X520 10Gb NIC. We disable hyperthreading and set the CPUs to a constant frequency of 2.6GHz to reduce the variance in benchmarking.

5.1 L2 Forward

We use DPDK's L2Forward example and benchmark it against a similar implementation in Rust. We vary the packet size from 100-1500 in steps of 100 with a batch size of 32 for both the implementations and sent packets at a rate of 10Gb/s (the line rate of the NIC). Since the CPU can easy saturate a 10Gb/s line while running at 2.6GHz, we also compare the performance at a frequency of 1GHz.

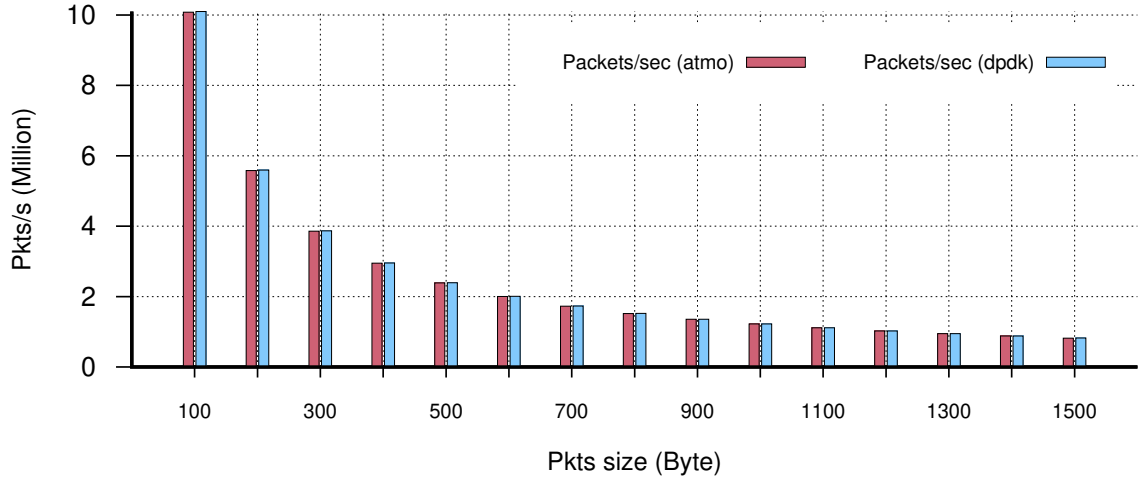

FIGURE 5.1: L2Forward throughput measured at 2.6GHz

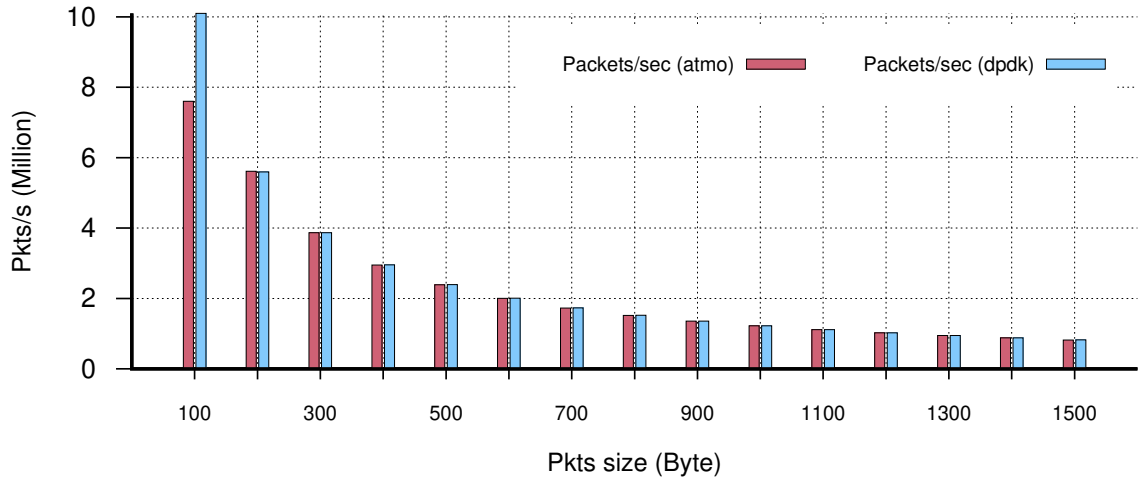

FIGURE 5.2: L2Forward throughput measured at 1GHz

5.2 Key-Value Store

To study the overheads of our network stack on a realistic workload, we implement an in-memory key-value store backed by a hash-table based on the Fowler-Noll-Vo algorithm. Processing a packet consists of parsing the request, performing an insert/fetch operation in the hash table, filling the response buffer queuing the packet in the send batch. Transmitting a packet takes fewer cycles than receiving because calls to `rx_batch` wait till the NIC is done writing packets into the DMA buffers while `tx_batch` simply holds the DMA buffers in a private queue and the NIC can write to them asynchronously. The next time `tx_batch` is called, the queue is cleaned and the buffers are returned as "free buffers".

capacity (power of 2)	operations (power of 2)	rx_tsc\pkt	process_tsc\pkt	tx_tsc\pkt
16	14	763	911	354
18	16	807	813	363
20	18	811	809	363
22	20	813	805	364
24	22	814	804	364
26	24	814	804	364
28	26	813	805	364

TABLE 5.1: Microarchitectural comparison of SFI and C on FFmpeg-x86 benchmark.

Bibliography

[1] Kevin Boos, Emilio Del Vecchio, and Lin Zhong. "A Characterization of State Spill in Modern Operating Systems". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: Association for Computing Machinery, 2017, 389–404. ISBN: 9781450349383. DOI: 10.1145/3064176.3064205. URL: https://doi.org/10.1145/3064176.3064205.

[2] Kevin Boos et al. "Theseus: an Experiment in Operating System Structure and State Management". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1–19. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/boos.