Rethinking Network Stack Design with Memory Snapshots

Michael Chan Heiner Litz David R. Cheriton
Department of Computer Science
Stanford University
{*mcfchan , hlitz , cheriton*}*@stanford.edu*

Abstract

Hardware virtualization is a core operating system feature. Network devices, in particular, must be shared while providing high I/O performance. By redesigning the network stack on a novel memory system that supports snapshot isolation, the operating system can effectively share network resources through the familiar socket API, enable zero-copy, reduce memory allocations and simplify driver communication with network interface cards. Starting with network I/O, we hope to further the discussion on hardware-software co-design to improve operating system architecture.

1 Introduction

Sharing hardware resources is the core functionality of operating systems. Each hardware device needs to be virtualized so that it can be simultaneously accessed by multiple threads of execution. A case in point is the network interface card, which must be shared by many threads with minimal impact on I/O performance.

Network stacks achieve resource sharing by continuously allocating and deallocating network buffers, and copying data between user and kernel space. Zero-copy techniques are application-specific [14][16] and have stringent memory access restrictions [17]. Kernel bypass reduces system-call overheads, but pushes the virtualization problem to a user-space library [13], or sacrifices resource sharing altogether [12]. An extreme case is providing per-thread hardware, thus virtualizing the network interface card (NIC) at the hardware level [8]. However, this does not match the ubiquitous sockets model, assumes tight coupling between threads using protocols incompatible with TCP/IP, and is not scalable to today's network applications involving thousands of loosely coupled threads.

Modern NICs and network stacks adopt a hybrid approach — NICs provide a small set of independent hardware resources. Each resource set is shared by a subset of threads, thus scaling out network I/O to multiple cores [6][11]. However, it has been shown that the NIC can be the bottleneck with increasing number of cores, even after extensive optimization of both the kernel and applications [1].

In this paper, we propose a novel approach to I/O device virtualization based on modifying the main memory architecture instead of providing explicit hardware virtualization support in the device hardware. Our approach, which targets, but is not limited to, network I/O, reduces network buffer management overhead, reduces the number of copy operations, simplifies NIC design and scales to a large number of cores. Our memory architecture provides isolation through immutability and snapshots. This allows secure access to shared resources, such as packet data, without the need for explicit memory pinning or data copying. This reduces the processing overhead for memory allocation and data copy, and the memory footprint of the network stack significantly. As memory regions are isolated by hardware, both software and hardware (NICs) can write to and read from memory without memory corruption concerns. Furthermore, this reduces the amount of overhead due to DMA address updates, and hence simplifies NIC DMA engines.

The paper is organized as follows. We elaborate on challenges in network I/O and describe the snapshot memory architecture in Section 2. We then present a new network stack design based on memory snapshots in Section 3, discuss the benefits in Section 4 and open issues in Section 5. While the paper is mostly focused on network I/O and network stacks, our hope is to spark discussions of how advanced memory-level features can be employed for high performance I/O, whether a redesign of other OS subsystems is valuable, and spur tighter interactions between designers of hardware and software are desirable [9].

2 Background

2.1 Challenges in Network I/O

Data copies: A network stack maintains multiple per-connection queues and per-NIC queues so that NIC resources can be shared by many concurrent threads. These queues are also necessary to absorb speed mismatches between user threads and the NIC, and are required to implement various protocol features, such as TCP in-order delivery. Queuing packets entails data copies to ensure memory isolation. Prior work shows that data copies incur a significant overhead [5].

Various zero-copy techniques have been proposed to minimize data copies. *sendfile()* supports transmission of data in the file cache, and hence is not applicable to data in user space. Some other solutions involve application-specific changes [14], and even putting the trusted application into the kernel [16]. Pinning user space buffers requires explicit buffer tracking from the application, imposes space limitations and often needs to resort to copying when the user thread is unable to keep up with received traffic [17]. Copy-on-write techniques allow kernel and user threads to share a packet buffer, however it operates on a coarse-grained page granularity, and are only efficient when the user thread rarely modifies the buffer after the *send()* call.

Therefore, the goal of our architecture is to minimize data copies without changing applications or sacrificing programmability.

Buffer allocations: Buffers — both packet data buffers and metadata buffers such as *sk_buff* — are continuously allocated and deallocated by the network stack. Buffer isolation among many concurrent user and kernel threads requires allocation of new buffers. For example, upon receiving new packets, the driver must allocate new buffers to refill the NIC DMA space, incurring significant overhead [7].

NIC and OS interactions: The operating system must ensure isolation between NIC DMA engine and other threads. As a result, the driver must create and maintain memory address mappings. The NIC obtains and stores memory mappings by reading DMA descriptors separately from packet data. NICs have limited descriptors, and if all descriptors are consumed, the NIC needs to drop packets.

2.2 The HICAMP Memory Architecture

We present an overview of the HICAMP architecture, a recently proposed memory system which provides snapshots, deduplicates memory and supports the familiar linear memory addressing scheme [3].

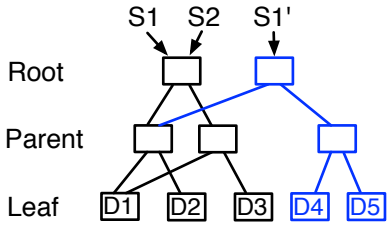

Figure 1: HICAMP memory lines, segments and segment mapping.

In HICAMP, physical memory is divided into equal-sized lines. Each memory line is addressed by a *Physical Line ID (PLID)* and is immutable — the content is written as part of line allocation and cannot be modified. Therefore, write operations lead to allocation of a new line instead of modifying the original content. A line may contain data or pointers to (i.e. PLIDs of) other memory lines. Application objects are accessed via memory segments, which are represented as a directed acyclic graph (DAG) of memory lines. Figure 1 provides an example of three HICAMP segments. The leaf lines of the DAG contains object data. Segment *S1* is mapped to the root line of the DAG containing data *D1, D2, D1, D3*. The DAG is augmented with parent lines as data is appended to the segment. HICAMP creates snapshots of objects by generating a new segment and mapping it to the same root line, effectively copying the original segment. Segment *S2* is thus a snapshot of the object. When *S1* is overwritten (becoming *S1'*), the data accessed through *S2* remains unchanged. Moreover, note that HICAMP automatically deduplicates memory at the line level. Deduplication occurs both within a DAG and across DAGs at all line levels by letting multiple parent lines point to the same data line.

The mapping from segment to root line is stored as a simple two-column translation table called the *segment map* stored in main memory and can be cached in a TLB-like hardware structure. Creating an object snapshot is very cheap, because it simply entails duplicating a mapping entry in the table, and the memory system automatically performs fine-grained copy-on-write in hardware. Each line has a reference count, which is incremented or decremented whenever a PLID reference to the line is established or removed respectively. When the count reaches zero, the line is deallocated and made available for writing again.

HICAMP memory is integrated into the memory hierarchy and virtual address translation, as illustrated in Figure 2. The process address space consists of two partitions — conventional and HICAMP space. Accesses to

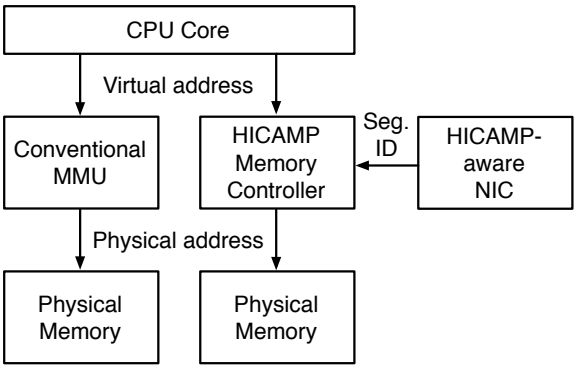

Figure 2: Memory system with HICAMP extensions.

virtual addresses in the HICAMP partition are directed to the HICAMP memory controller. The controller then translates virtual addresses into physical addresses by extracting the segment ID and leaf offset from the virtual address.

3 Redesigning the Network Stack

Packet transmission and receipt using HICAMP memory operates as follows. A user application prepares a packet by creating a segment comprised of immutable lines. Instead of copying the entire packet into the kernel space it only passes a reference to the segment. The kernel network stack prepends header information and finally passes an updated reference to the NIC without ever copying payload data. Reference passing involves creation of a snapshot, hence user and kernel space software is allowed to overwrite packet buffers immediately after passing them to the next layer as the original content is never modified due to immutability. Our technique combines asynchronous zero-copy transmission with a clean and synchronous socket interface, unburdening the programmer of pinning memory, performing synchronization or checking whether buffers can be reused. Similar techniques are used on the receive side. We now elaborate on packet representation, transmission and receipt.

3.1 HICAMP Network Packets

Packets are represented as HICAMP segments. Figure 3 shows how a transmit packet is created. Initially, the payload being sent is appended at a certain leaf offset in the segment (Figure 3a). The offset is set to provide enough space for protocol headers and NIC descriptor metadata. The header space is initialized with zeroes and gets populated while traversing the network stack as shown in Figure 3b. HICAMP snapshots enable concurrent changes

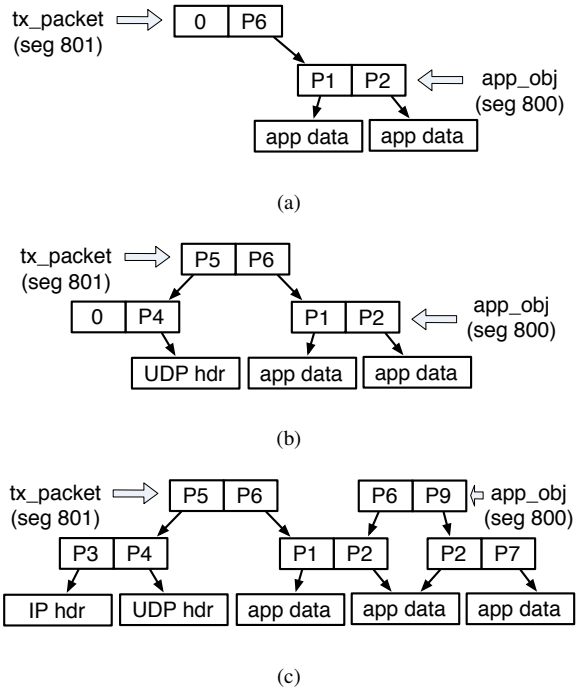

Figure 3: Example of creating a packet for transmission.

to segments without compromising isolation. While protocol processing is taking place, modifications to the payload through the application object segment do not affect the payload of the packet segment due to immutability of lines (Figure 3c).

3.2 Network Packet Transmission

The redesigned transmission path is illustrated in Figure 4. It has two main components: Interactions between user-space and kernel-space as well as interactions between the driver and the NIC.

User and kernel interactions: The user process creates an application-level object as HICAMP segment 9000, which is mapped to the root PLID 1. The object is transmitted through the *send()* system call. The kernel creates a snapshot of the object and appends it to segment 30. This new segment, called the *stack segment*, is passed through the TCP/IP protocol stack. The various layers append protocol headers to the stack segment, as depicted in Figure 3.

Driver and NIC interactions: The device driver further writes some metadata to the stack segment. This metadata is used to communicate control information to the NIC. For example, this could include offsets of the IP and TCP checksums in the packet and the

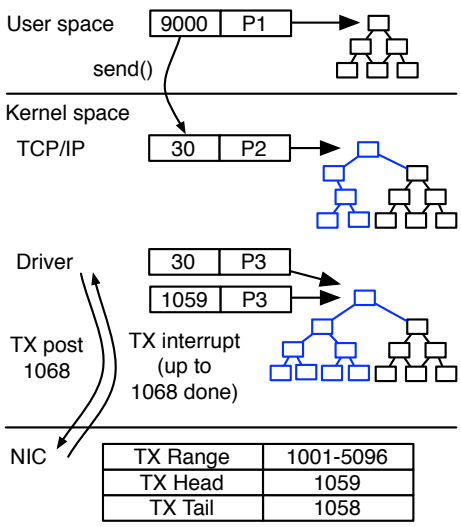

Figure 4: Using memory snapshots for zero-copy packet transmission.

number of MSS-sized segments for the hardware TCP segmentation engine.

The NIC accesses memory with DMA as in a conventional architecture. However, it uses segment IDs instead of physical addresses for DMA operations (Figure 2). The transmit DMA state is represented by 4 segment IDs. Two segment IDs define a contiguous range of segments for DMA reads from the HICAMP memory controller. The range creates a TX DMA ring similar to the conventional TX descriptor ring. However, unlike contemporary DMA, metadata is embedded within the segments instead of being stored in a separate descriptor ring. The NIC also maintains two segment IDs, TX Head and TX Tail. Segments in the range [TX Tail+1, TX Head-1] are ready for transmission by the NIC, whereas segments in the range [TX Head, TX Tail] are owned and being processed by the operating system.

Once a stack segment is ready for transmission, a snapshot is created by the driver. The next segment in the OS-managed part of the TX segment ring (TX Head) is used for the snapshot mapping. For example, in Figure 4, segment 1059 is used. The driver may accumulate multiple segments before starting the DMA process. In the example, the driver prepares 10 segments before issuing a TX post to the NIC, indicating that the NIC should read and transmit segments from TX Head up to segment 1068. The NIC requests for these segments from the HICAMP memory controller and transmits them. After that, the NIC updates TX Head and TX Tail to 1068 and 1069 respectively, and issues a TX interrupt. The interrupt handler reads the status code, discovers that all segments have been transmitted, and recycles segments for future transmissions. Finally, the driver updates its copy of TX Head and TX Tail by reading the NIC's values.

3.3 Packet Receipt

The packet receive process follows a similar pattern. The NIC deposits received packets into the available RX segments. Each segment contains receive-specific metadata followed by the packet. The NIC then issues an RX interrupt. The driver reads segments from the OS-managed part of the RX ring. For each segment, a snapshot is taken, which produces a stack segment. The stack segment is passed up the protocol stack. The RX ring segments are now available for packet receipt. The driver informs the NIC of the newly processed RX range.

The *recv()* system call creates a snapshot of the stack segment to produce a user-owned segment for application processing. The stack segment can then be recycled for future receipts.

4 Benefits

Zero copy with *send()* and *recv()*: Packet snapshots eliminate data copies between kernel and user space, because application payload is passed as segment references. Because HICAMP is integrated into the virtual memory architecture, one can provide an enhanced C library which extends *malloc()* to allocate virtual addresses backed by HICAMP segments. *send()* and *recv()* can also be patched to pass segment references to the kernel instead of copying. This way, application developers can benefit from zero copy with the familiar POSIX sockets API. In contrast, buffer sharing schemes such as IO-Lite [10] and fbufs [4] require all application objects to be allocated within special IO buffer data structures, which increases programming complexity.

Reduced per-packet memory overhead: Using hardware-provided memory snapshots, the network stack needs not explicitly allocate memory to isolate user and kernel data. Furthermore, in packet receipt, after delivering a segment up the stack, the driver needs not reallocate worst-case-sized RX buffers.

System-wide memory deduplication: The HICAMP controller deduplicates memory at line level. While zero-copy reduces redundancy by reusing the same buffer, hardware deduplication further exploits redundancy between all buffers. One can further deduplicate data in the last-level cache, hence effectively increasing the size of both main memory and cache. Some common datasets such as Web pages and images exhibit significant deduplication potential [3]. Buffer sharing schemes based on paged memory [2][4][10] are

unable to exploit this redundancy.

Simplified NICs: The NIC needs less hardware resources for DMA state, because the head and tail pointers suffice for memory addressing. The simplified DMA engine could support a larger RX ring to better absorb traffic bursts. Moreover, the NIC issues a single bus transaction to access a range of segments containing both descriptor metadata and payload. Reducing bus transactions further amortizes interconnect overhead.

Efficient memory protection: Snapshots based on HICAMP segments are guaranteed by hardware to be completely isolated from other segments. It does not require page table modifications and TLB shootdowns as in existing zero copy schemes [2][10]. Optimizations have been proposed to lower mapping overhead, but at the cost of reduced memory protection and puts extra burden on developers to be cognizant of buffer access control [4]. In contrast, memory protection in HICAMP is implicit and efficient. Moreover, in HICAMP-based DMA, NICs must access memory through segments allocated in the RX and TX rings during device initialization. Indirection through segments effectively implements memory protection provided by IOMMUs.

5 Open Issues

Space tradeoff: Previous studies have shown that duplicates in application data offset the increased memory consumption by DAG structures [3][15]. In network I/O, HICAMP segments avoids memory allocations via conventional pages, so there are fewer page metadata structures. However, the total network stack memory footprint also depends on the duplication degree in packet data. Moreover, because incoming traffic is uncontrollable, it is desirable to have sufficient memory to absorb receive bursts from the NIC. The operating system can explicitly track total memory consumption, and selectively deny/delay segment allocation requests to prioritize memory space for time-critical operations, such as receive-side DMA. To handle memory exhaustion, the operating system may swap segments into disk, but this raises issues on segment selection and performance overhead. Another option is to fallback to conventional memory, but this complicates the stack and NIC with two parallel memory management schemes. We believe a reasonable approach is to use HICAMP memory exclusively to reduce complexity, provision memory based on evaluation of various workloads, and delay non-urgent memory allocations. Overall, analysis of the overall deduplication degree and memory footprint is needed to determine the best operating point for network I/O.

Time tradeoff: Maintaining the DAG structure requires more memory accesses, which increases memory bandwidth overhead. The open question here is whether the extra overhead is a reasonable tradeoff for reduced number of software memory allocations. The deduplication degree again plays an important role in this tradeoff. If there is a high degree of duplication in packet data at the memory line granularity, then some packets can be allocated through pointer manipulations, saving memory accesses. The time-scale of duplication is also significant. For example, during packet RX, two packets can contain duplicated payload. However, if the earlier packet is completely processed and deallocated by the system before the latter packet arrives, then part of the latter's DAG structure is rebuilt unnecessarily. An optimization is to delay the deallocation of memory lines for packets to achieve better deduplication.

Impact on kernel subsystems: Because the network stack uses HICAMP memory exclusively for packet data, interfaces with other kernel subsystems will need to be updated. For example, *splice()* will require copying from HICAMP memory to conventional memory so that the file system can operate on the data. A translation layer that is knowledgeable of both HICAMP and conventional memory allocators in the kernel is needed. Alternatively, HICAMP memory can be employed for file I/O as well. For instance, in a *splice()*-supported file receive process, the data segment is directly passed to the segment-based file system code. The segment is then pushed down through the block layer to the disk driver. Furthermore, the file cache could be layered on top of HICAMP to exploit line-level redundancy in file contents. More generally, one can apply either technique based on evaluating tradeoffs between performance and complexity of incorporating HICAMP into a subsystem.

6 Conclusions

Hardware virtualization is a core operating system feature. High-performance device sharing is desired especially for network I/O. In this paper, we posit that network I/O perforamnce can be enhanced with a radical change in memory architecture. A network stack redesign based on memory snapshots and dudplication realizes zero-copy, reduced memory allocations and simplifies drivers and NICs, while retaining the familiar socket API. We believe architectural innovations are crucial to further improve operating systems. By identifying various open issues in network I/O, we hope to contribute to discussions of hardware-software codesign for operating systems.

References

[1] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, OSDI'10, USENIX Association, pp. 1–8.

[2] BRUSTOLONI, J. C., AND STEENKISTE, P. Effects of Buffering Semantics on I/O Performance. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, ACM, pp. 277–291.

[3] CHERITON, D., FIROOZSHAHIAN, A., SOLOMATNIKOV, A., STEVENSON, J. P., AND AZIZI, O. HICAMP: Architectural Support for Efficient Concurrency-safe Shared Structured Data Access. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, ACM, pp. 287–300.

[4] DRUSCHEL, P., AND PETERSON, L. L. fbufs: a High-bandwidth Cross-domain Transfer Facility. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, ACM, pp. 189–202.

[5] FOONG, A. P., HUFF, T. R., HUM, H. H., PATWARDHAN, J. R., AND REGNIER, G. J. TCP Performance Re-visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '03, IEEE Computer Society, pp. 70–79.

[6] HAN, S., MARSHALL, S., CHUN, B.-G., AND RATNASAMY, S. MegaPipe: a New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, USENIX Association, pp. 135–148.

[7] LIAO, G., ZNU, X., AND BNUYAN, L. A New Server I/O Architecture for High Speed Networks. In *Proceedings of the 2011 IEEE International Symposium on High Performance Computer Architecture*, HPCA '11, IEEE Computer Society, pp. 255–265.

[8] LITZ, H., FROENING, H., NUESSLE, M., AND BRUENING, U. Velo: A Novel Communication Engine for Ultra-Low Latency Message Transfers. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on* (sept. 2008), pp. 238 –245.

[9] MOGUL, J. C., BAUMANN, A., ROSCOE, T., AND SOARES, L. Mind the gap: Reconnecting Architecture and OS Research. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'11, USENIX Association, pp. 1–5.

[10] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. IO-lite: a Unified I/O Buffering and Caching System. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, USENIX Association, pp. 15–28.

[11] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, ACM, pp. 337–350.

[12] RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, USENIX Association, pp. 101–112.

[13] SHALEV, L., SATRAN, J., BOROVIK, E., AND BEN-YEHUDA, M. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10, USENIX Association, pp. 61–74.

[14] SONG, X., SHI, J., CHEN, H., AND ZANG, B. Revisiting Software Zero-copy for Web-caching Applications with Twin Memory Allocation. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, USENIX Association, pp. 355–360.

[15] STEVENSON, J. P., FIROOZSHAHIAN, A., SOLOMATNIKOV, A., HOROWITZ, M., AND CHERITON, D. Sparse Matrix-vector Multiply on the HICAMP Architecture. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, ACM, pp. 195–204.

[16] STUEDI, P., TRIVEDI, A., AND METZLER, B. Wimpy Nodes with 10GbE: Leveraging One-sided Operations in Soft-RDMA to Boost Memcached. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, USENIX Association, pp. 347–354.

[17] ULRICH DREPPER. The Need for Aynchronous, Zero-Copy Network I/O: Problems and Solutions. In *Ottawa Linux Symposium* (2006), pp. 247–260.