FShm: High-Rate Frame Manipulation in Kernel and User-Space

John Giacomoni[†], John K. Bennett[†], Antonio Carzaniga[†‡],
Manish Vachharajani[†], and Alexander L. Wolf[†‡]
† *University of Colorado at Boulder*
‡ *University of Lugano*

University of Colorado at Boulder
Technical Report CU-CS-1015-06
October 2006

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309-0430

FShm: High-Rate Frame Manipulation in Kernel and User-Space

John Giacomoni[†], John K. Bennett[†], Antonio Carzaniga[†‡],
Manish Vachharajani[†], and Alexander L. Wolf[†‡]
† *University of Colorado at Boulder*
‡ *University of Lugano*

Abstract

The high performance, low cost, and flexibility of commodity hardware systems make them appealing for network processing applications. However, the standard software architecture of such systems imposes significant limitations. At high rates (e.g., Gigabit Ethernet) and small frame sizes (64 byte) each frame must be processed in less than 672 ns. System calls, synchronization, and memory latencies can dominate this processing time. Despite significant effort to remove this overhead, we are aware of no general purpose mechanism that can handle this load on commodity hardware.

This paper describes the frame-shared-memory architecture (FShm), a general-purpose software architecture for processing network frames on commodity multiprocessor hardware. FShm supports *kernel-* and *user-space* processing at Gigabit Ethernet rates by increasing throughput, without reducing the per-frame processing time, by pipelining work across multiple processors. FShm can generate, capture, and forward frames at the theoretical maximum rate on a Gigabit Ethernet network for all frame sizes greater than 96 bytes, and at 95% of maximum for the 64 byte minimum frame size (the limit of the tested hardware).

1 Introduction

Commodity hardware systems are an appealing choice for network processing applications that demand flexibility and low cost. Unfortunately, the software architecture of these systems imposes a number of limitations on the types of services that they can provide, especially for frame processing tasks. For example, consider a Network Intrusion Detection (NID) system designed to identify malicious traffic on a network. Since malicious traffic may occur at the Ethernet frame level in any frame, such a system must be able to handle data at the frame level, for all valid frame sizes, at the wire rate.

For larger frame-sizes, there are a number of software techniques to process data arriving over a very high-bandwidth link [5, 12, 18], such as a Gigabit Ethernet. Unfortunately, as the frame size decreases, the total time available to process each frame also decreases. For example, for a Gigabit Ethernet link saturated with 64 byte frames (the minimum frame size), one has only 672ns to process a frame [24]. However, on an AMD Opteron or Intel Xeon, the overhead for a single system call (excluding any real work done by the OS) ranges from 171ns to 534ns; a single lock-unlock pair may cost as much as 354ns. These overheads are a significant fraction of the total frame processing time. While some applications can cope by dropping frames, in other applications such as NID, a frame drop may be considered a definitive application failure, since the dropped frame may contain an attack which would be missed by the NID system.

Modern high-bandwidth processing techniques eliminate almost all frame-processing overhead, either by avoiding locks and system calls altogether, or by aggregating data transfers to amortize the cost of these overheads across many frames. Despite such effort, no existent frame processing technique has been able to demonstrate the ability to process large numbers of small frames at Gigabit Ethernet rates on commodity hardware.

This paper presents the Frame-Shared-Memory (FShm) software architecture for high-rate frame processing. FShm allows for processing of small frames at Gigabit Ethernet rates in user-space with no dropped frames. FShm accomplishes this by using multiple processors to build a processing pipeline that improves throughput, but does not decrease the available per-frame processing time for applications. In fact, by utilizing additional processors, FShm may increase the per-frame processing time for applications. This allows FShm to scale to faster future networks by using the extra cores that future commodity multi-core processors will provide. This paper describes experiments that demonstrate the transmission, reception, and forwarding of Gigabit Ethernet frames at the theoretical maximum with no drops for frames 96B and larger, and at at 95% of the theoretical maximum wire-rate (the most the hardware can generate) with no drops for 64B frames. Furthermore, we demonstrate that FShm allows user-space threads to perform sophisticated processing.

The remainder of this paper is organized as follows. Section 2 precisely describes FShm's design goals and then discusses system constraints that make achieving these goals difficult. Section 3 then presents the design of the FShm pipeline and discusses how FShm overcomes each of the bottlenecks in Section 2. Section 4 evaluates how well FShm meets its design goals by measuring its performance at Gigabit rates at a variety of frame sizes for a range of tasks. Section 5 presents related work not covered elsewhere in the paper. Section 6 concludes.

2 Design Criteria and Constraints

This section defines the the design goals for FShm and discusses the bottlenecks present in commodity systems.

2.1 Desired Features

FShm was designed to satisfy the following two criteria:

1. FShm must provide the ability to transmit, receive, and forward packets at close to the wire-rate (at Gigabit Ethernet speeds, for the purposes of this paper) without increasing the drop rate imposed by the underlying hardware (ideally dropping no frames).

2. FShm must have overhead that is small enough so real work can be accomplished by the application when dealing with a network saturated with small frames. A system that consumes all per-frame processing time simply to deliver data to and send data from an application is nearly useless.

In addition to high performance, a general-purpose framework is desirable. Exactly what is considered general purpose is clearly influenced by the targeted application. However, certain properties are required by a large number of network applications, and FShm has been designed to support these. The additional criteria imposed by the desire for a general purpose system are as follows:

3. FShm must be able to process *every* frame on the incoming network interface. To understand this requirement, consider applications such as network intrusion detection (NID). Since any frame may contain an attack, dropped frames may result in missed attacks. If a system has trouble at high-rates, attackers can evade detection simply by sending a flood of small frames. It is easy to imagine additional applications that might monitor every frame.

4. FShm should support both in-order and out-of-order frame delivery. Consider a router application. This application will likely want to process frames out of order, to boost performance. However, a NID system may require in-order delivery of frames.

5. FShm should support user-space processing of frames since complex applications, such as NID or content-based routing, may involve sophisticated user-specified code written in high-level languages with an eye towards portability.

6. FShm should provide a mechanism to trade-off latency for expanded processing time without sacrificing throughput, since some applications require fairly complex operations on a per-frame basis.

2.2 Network Imposed Constraints

The first constraint faced by any frame processing system targeting wire-rate processing with no drops is the frame arrival rate (or the inter-frame arrival period). Consider a saturated Gigabit Ethernet link. On such a link, the inter-frame arrival period is given by

$$T = \frac{\text{InterframeGap} + \text{Pre\&Postamble} + \text{FrameSize}}{\text{Bitrate}}$$

where the $\text{InterframeGap} \geq 96\,\text{bits}$, $\text{Pre\&Postamble} = 64\,\text{bits}$, and FrameSize is the frame size measured in bits (including the $32\,\text{bits}$ of CRC). Setting the $Bitrate = 1 \times 10^9\,\text{bps}$ and the $\text{InterframeGap} = 96\,\text{bits}$, we can readily find the maximum frame rate at different frame sizes. For example, with 64B frames, there are 1,488,095 frames per second (fps) implying the system needs to be ready for a frame every 672 ns. With 256B frames there are only 452,951 fps or a frame every 2207 ns. On a 2GHz machine (e.g., the AMD Opteron on which FShm was evaluated) this is the difference between 1338 processor cycles and 4397 cycles.

The smaller the frame size, the more difficult it is to support wire-rate, since the frame-rate is considerably higher, even though the total bandwidth utilized is less due to constant per-frame overhead on the network. Contrast the $1,488,095\,\text{fps}$ at $761\,\text{Mbps}$ for 64B frames versus the meager $81,274\,\text{fps}$ at $987\,\text{Mbps}$ for 1518B frames. Thus, the subsequent discussion on system bottlenecks focuses on the 64B frame-size with the implicit understanding that if a system can handle 64B frames it can easily handle larger frame sizes.

2.3 Operating System Bottlenecks

To understand the Operating System bottlenecks in frame processing, it is useful to examine the application flow

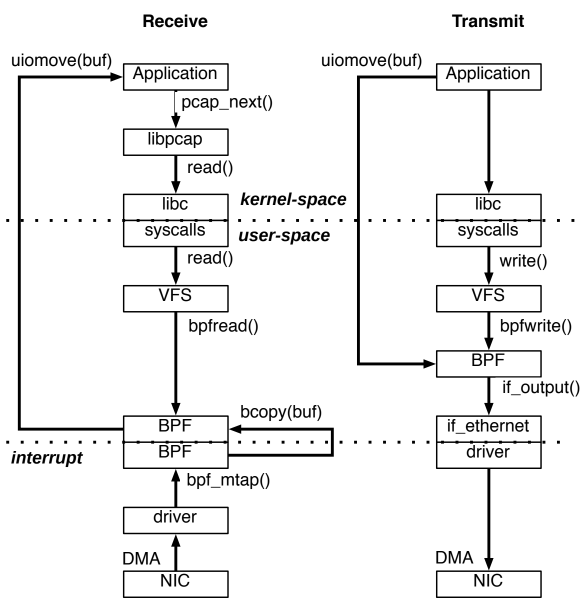

Figure 1: Call chain for the Berkeley Packet Filter

through both kernel and user-space for a canonical network processing system. Figure 1 shows the flow of the send and receive operation in the Berkeley Packet Filter (BPF) [23] network application as implemented in FreeBSD, which is representative of common user-space network processing. Here, the application must go through six or more levels to send or receive a packet. On the receive side libpcap[20] is typically used to read from the BPF interface and provide a cross-platform frame interface. Libpcap invokes the read system-call on the BPF device node. In the kernel, the VFS layer translates this call to a read from the kernel-space BPF buffer. This read operation invokes uiomove which then copies the buffer into user-space. At interrupt time, when the DMA transfer of new frame data from the network interface card (NIC) is complete, the BPF layer copies the data from the driver into the BPF buffer. Send operates in reverse order. The application calls write on the network device, which invokes the kernel. The VFS layer translates this into a call to the BPF subsystem which uses uiomove to copy the user data into kernel-space. From here, the data is passed to the Ethernet driver which then DMAs the data to the NIC.

These mechanisms require several copies (the bcopy and the uiomove), a system-call, and passage through the VFS layer. Furthermore, since the BPF buffers are accessed simultaneously by the Ethernet driver and the BPF kernel code, a lock must be acquired and released before the BPF buffers can be accessed. The three main Operating System bottlenecks in this process are the system calls, lock acquisition, and the VFS layer. These are

# VCPU	# Thr.	System Calls (ns)	ioctl (ns)
0	1	171	538
1	1	851	1826
1	2	1334	1674
2	2	911	1284
2	3	912	1223

Note: For VCPU=0, no scheduler activations are used; scheduler activations result in reduced performance even for single thread tests.

Table 1: System-call and ioctl costs on an AMD Opteron.

discussed below. (Overhead due to copies, which is primarily memory overhead, and I/O costs are covered in Section 2.4.)

2.3.1 System Calls

For most applications, the cost of system calls are irrelevant as they occur relatively infrequently compared to other application operations. However, when handling the frame rates described above, they quickly overwhelm the system's ability to process frames resulting in a significant number of dropped frames. Table 1 quantifies the costs of performing system calls on a dual processor dual core 2.0GHz AMD Opteron 270, running FreeBSD 5.5 (the fastest configuration we had available, according to our measurements). The "# VCPU" column indicates the number of processors available to the application (as defined by Scheduler Activations [1]), and the "# Thr." column indicates the number of concurrent threads in the process for which the system-call time was measured. Notice that the Opteron system is not able to perform even a single system call and remain within the 672 ns service time for a single 64B frame on a saturated Gigabit Ethernet link, assuming a multi-threaded environment. This overhead is due to the additional complexity of the user-space scheduler provided by scheduler activations.

Most recent prior work uses shared memory to transfer data between the NIC device driver and the network processing application to avoid the cost of system calls[29, 21]. As Section 3 describes, this is the approach used by FShm to eliminate system calls for the transfer of data between different contexts, be it between a kernel and user context or multiple user contexts (i.e., processes).

2.3.2 Locking Mechanisms

In order to prevent concurrent access to a single buffer (resulting in corrupted data), most production operating systems use mutexes (also known as locks). Table 2 enumerates the costs of mutex operations in kernel- and user-space for the same AMD Opteron described earlier. The

Lib	Lock (ns)	Unlock (ns)	Lock & Unlock (ns)
pthread	75	78	159
kernel	28	19	27

Table 2: Mutex operation costs for an AMD Opteron

minimum cost of passing a variable between two threads, assuming no cache misses or contention on the locks, is double the cost of a single lock/unlock pair since both the reader and writer must lock the data structure when performing the respective operation. This implies that on the order of 318 ns (for user↔user communication) or 54 ns (for kernel↔kernel/user) would be spent passing a single variable out of a budget of 672 ns for 64B frames; this is intolerable in the former case and less than ideal in the latter. Note that these numbers are best-case because the mutex variable is already in the processor cache.

One could aggregate writes to amortize the cost of a lock-unlock pair. However, the costs of such aggregation must still be addressed. Section 3 describes how FShm eliminates the need for explicit locking by utilizing concurrent lock-free queues and leverages the cache coherency manager to implicitly achieve the locking needed to ensure data consistency.

2.3.3 VFS Layer Overhead

The indirection of reads and writes through the VFS (filesystem) layer of the kernel incurs the largest overhead. Table 1 shows the cost of a null ioctl call (thus capturing just the overhead of the system-call and VFS layer) for the same Opteron system. We can see that the cost of an ioctl can be up to triple that of a plain system call. Recall that system calls alone overwhelmed the 672 ns processing windows for 64B Gigabit Ethernet frames, passing through the VFS layer is clearly intolerable for FShm. Section 3 describes how FShm's shared-memory architecture avoids both the system-call and VFS layer costs.

2.4 Hardware Bottlenecks

While the hardware plays a prominent role in the cost of the various OS bottlenecks described above, there are other potential bottlenecks in the hardware that are more difficult to control. Of these, the overall system I/O architecture and memory subsystem play the most significant role. Since it is difficult for software techniques (such as FShm and much of the prior work) to mitigate these bottlenecks, one should bear in mind that they may ultimately limit the performance of *any* system. Fortunately, FShm is still able to handle 64 byte frames at Gigabit Ethernet rates for the Opteron system on which it was evaluated (See Section 4).

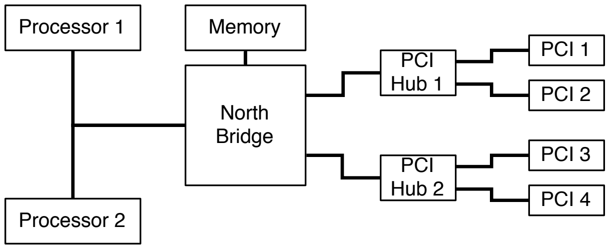

Figure 2: Host Bus Architecture

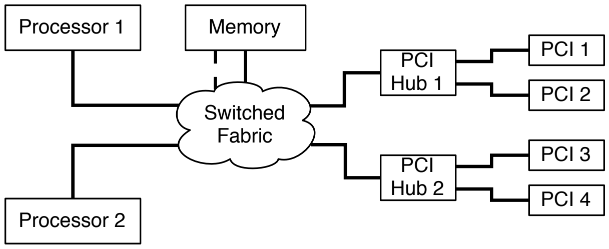

Figure 3: Switched Architecture

2.4.1 System Architecture

To understand the potential bottlenecks in the system architecture, consider Figure 2 and Figure 3. In the host bus architecture (Figure 2), the first potential bottleneck is the shared communication bus between the two processors. Since access to memory, access to snoop results from other processors, and access to the PCI bus all contend for the same interface, it can become a bottleneck.

Even if the processor front-side bus is not a problem, the internal organization of the North Bridge can limit performance. For example, it may be the case that a PCI device on one bus cannot DMA to memory at the same time that a processor wishes to deliver a command to the other PCI bus. Other issues in the North Bridge design, such as interrupt scheduling and command latency, can also have tremendous impact on total achievable throughput to NICs on the PCI bus.

Consider the data in Table 3 which compares the num-

System	NIC	PCI Bus	Frame Size	Max Frames/s
1.0 GHz PIII	Intel 82545 GM	64-bit 66 MHz	64B	811 kfps
2.66 GHz P4 Xeon	Intel 82545 GM	64-bit 133 MHz	64B	723 kfps

Table 3: Throughput on a Pentium III & Pentium 4 Xeon.

ber of minimum size Ethernet frames that can be generated on a Pentium III system and Pentium 4 Xeon system with Linux pktgen [27]. Notice that the Pentium III system has a slower PCI bus, is a slower processor, and yet can place frames on the network faster than the Pentium 4 Xeon system (our numbers confirm this). However, the Pentium III system contains a high-end server motherboard that utilizes the ServerWorks HE-SL chipset, whereas the Pentium 4 Xeon has a standard motherboard. Here, the chipset makes all the difference.

The switched system architecture shown in Figure 3 is an improvement over the bus-based architecture in that multiple independent data transactions need not interfere. For example, depending on the network topology in the switched cloud, it is possible for a processor to access main-memory while another processor services a cache snoop request. This reduces the contention for limited system resources, however, bottlenecks may still exist since the switched interconnect network is not a full crossbar. The AMD Opteron system on which FShm is evaluated is a switched architecture.

2.4.2 Memory Latency

Another major obstacle to high-performance frame processing is the discrepancy between processor clock rates and main memory. It can take hundreds of CPU clock cycles to fetch a single cache line from the memory subsystem (translating to 50-100 ns). Furthermore, modern system architectures make this penalty difficult to avoid. DMA transfers from commodity NICs always write to main memory forcing the processor to fetch data from main memory. Since cache line size is on the order of frame size for small frames, each frame requires a main memory transaction. Some of this overhead can be hidden through prefetching [7] and other techniques, but the tight time bound for processing 64B frames at gigabit rates, limits the effectiveness of these techniques.

3 FShm Design

Meeting the design goals enumerated in Section 2.1 is difficult given the overheads in the existing frame socket interfaces. Previous work has reduced the per-frame costs by minimizing this overhead via aggregated transfers or "safe" zero-copy semantics. While these techniques minimize almost all the overhead, the per-frame non-application processing time still exceeds the 672 ns available for 64B frames on a saturated Gigabit Ethernet link.

This section describes the design of the FShm architecture. FShm is capable of processing 64B frames on a saturated Gigabit Ethernet link with no drops. FShm

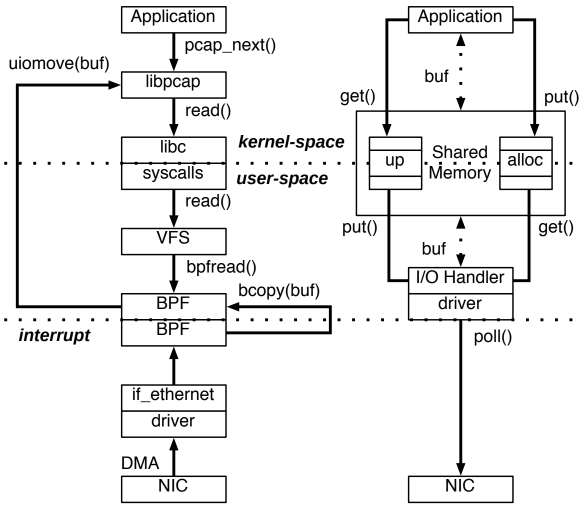

Figure 4: BPF Receive (left) and FShm receive (right)

concedes that the prior work has done as much as possible to reduce per-frame processing overhead on a single processor system. Instead of targeting overhead, FShm uses the concept of pipelining (extremely common in hardware design) to distribute the work for each frame across multiple processors in software. This allows increased frame throughput without the need to reduce per-frame processing overhead. In fact, by increasing the number of pipeline stages, FShm can expand the available time to process each frame by increasing frame latency, but still maintaining frame throughput.

To understand the design of FShm, this section first discuss why it is unlikely that per-frame processing time can be reduced on a single processor machine. The section then introduces the pipeline organization used in FShm to improve throughput and increase available processing time, even though per-frame processing overhead is not reduced. The section concludes by describing how the inter-stage (i.e, processor to processor) communication mechanism is designed to avoid lock-unlock and system-call overhead.

3.1 Optimizing the Frame Socket Interface

To understand why single-processor per-frame processing overhead is unlikely to be reduced, it is instructive to examine the bottlenecks in the frame socket interface (described earlier in Section 2) and examine how prior work mitigates these costs. To recap, the critical bottlenecks are the system calls, navigating the VFS layer, the double copies (i.e., the bcopy and the uiomove), and the locking between the interrupt handler and the high-level of the kernel, as shown in Figure 4.

Implementations of the BPF attempt to minimize the impact of the copies, locking, and system calls by aggregating multiple frames worth of data into a single buffer and returning the entire block in a single system call. By using two buffers, the lock contention between the interrupt handler and the kernel is minimized since the lock only needs to be acquired when swapping the buffers or copying the contents to user-space. Unfortunately while all the bottlenecks are minimized, they are not eliminated. By design the bcopy in the interrupt handler cannot be eliminated or minimized as the system is designed to forward the frame to the network stack in addition to capturing it. Monitor mode prevents the frame from traveling the network stack, however, the bcopy is still necessary to maintain the semantics of the higher-level API.

Other prior work focuses on eliminating the overhead of the uiomove operation that copies frames from kernel-to user-space as copies can be a significant bottleneck [6, 5, 18]. These techniques provide loose copy semantics, which is all that is necessary for *high-bandwidth* (vs. high frame-rate) applications. Additional work eliminated the copies by memory mapping a shared region of kernel memory into the application's address space [21]. These solutions minimize or eliminate the cost of copying but retain the overhead of the system calls and the VFS layer.

Alternative approaches focus on minimizing the kernel overhead by employing NICs capable of DMAing frames directly into the address spaces of interested applications [12, 30]. This is done by mapping the NIC's buffer rings directly into an application's address space [9] or by permitting user-space DMA.

The approaches described above completely eliminate buffer copying via shared pages across the user-space/kernel-space boundary, and completely eliminate system calls and VFS overhead by DMAing directly into user-space. However, none techniques can eliminate the overhead of DMA transfers and other memory accesses. Furthermore, none of these techniques allow Gigabit rate processing of small frames, even though overhead is minimized. As a result, the real non-application overhead involved in processing frames on commodity hardware exceeds the service time for small frames at gigabit rates.

Therefore, we believe it is necessary to develop techniques that can improve throughput without requiring further reductions in the per-frame processing time (including overhead).

3.2 The FShm Pipeline

As shown in Figure 4, the design of FShm eliminates the previously described overheads. The key to FShm's ability to process small frames at gigabit rates comes from its use multiple processors (or multiple cores on multicore systems) to pipeline per-frame work. This improves frame throughput without sacrificing latency

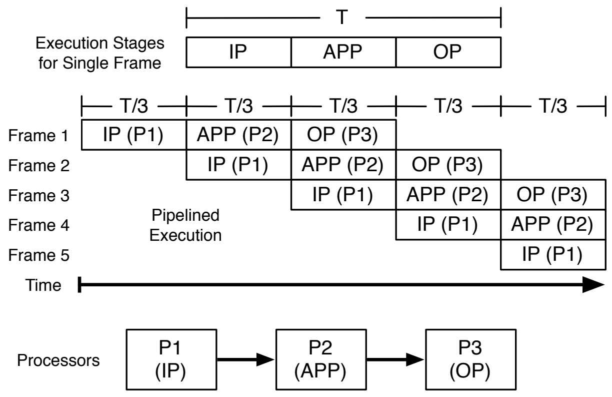

Figure 5: The Basic FShm Pipeline

Figure 5 depicts an overview of the FShm processing pipeline. From the diagram, we see that each frame takes T nanoseconds to process. From the previous discussion, we know that $T \geq 672ns$, the inter-frame arrival period for 64-byte frames on a saturated Gigabit Ethernet link. While we cannot reduce this processing time, it is possible to improve the throughput of processing by using multiple processors as shown in the figure.

In the general case, the processing of each frame is divided into at least three stages, Input processing (IP), Application processing (APP), and Output processing (OP). There are degenerate cases where there is no IP phase (e.g., frame generation) and where there is no OP phase (e.g., frame capture). Each stage is executed as a thread on a separate processor. Ignoring communication overhead, and assuming an ideal division of labor, we see that the processing for each frame can be divided into stages with duration $T/3$ nanoseconds. Thus, each frame still takes T nanoseconds to go from input to output. This agrees with the earlier assertion that per-frame processing time cannot be reduced. However, by dividing the work across 3 processors, it is possible to initiate processing on a new frame every $T/3$ nanoseconds, tripling the frame throughput. Because there is still only one processor handling input for the NIC(s), and another for the output, no mutual exclusion is needed to manage the resources because there is no contention. In short, we parallelize frame processing without introducing contention.

In reality, communication overhead is introduced by splitting work into stages that run on separate processors. Furthermore, the IP, OP, and APP stages may not each take a third of the total time. Typically, the APP stage will be much more time consuming than IP and OP. However, even in this case throughput still increases, though

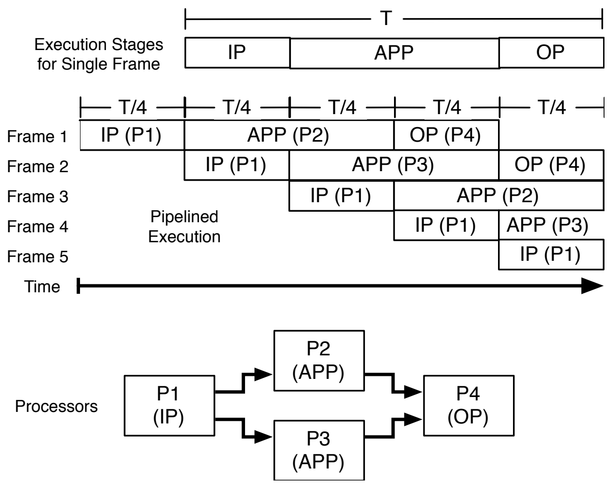

Figure 6: The FShm Pipeline with multiple APP Units

```
put_nonblock(...) {
  if (NULL == queue[head])
  {
    queue[head] = ptr;
    head = NEXT(head);
  }
}

get_nonblock(...) {
  if (NULL != queue[tail])
  {
    ptr = queue[tail];
    queue[tail] = NULL;
    tail = NEXT(tail);
  }
}
```

Figure 7: CLF Queue: Put_nonblocking

not optimally. It is possible to further improve throughput in this unbalanced scenario by utilizing additional processors to allow multiple APP stages to run in parallel. Effectively, this expands the amount of processing time per-frame without affecting throughput. Figure 6 shows such an example where the APP stage takes twice as long as either IP or OP and cannot be divided amongst processors. Here by utilizing a single extra processor, it is still possible to quadruple the frame throughput versus the original single processor solution. Again, this happens without shortening the per-frame processing time.

This pipeline structure and careful implementations of inter-process communication, allows FShm to process frames at Gigabit Ethernet rates, even for small frames. With shared memory inter-process communication between kernel and user space, pipeline stages can occur in either in kernel-space or in user-space. In FShm, we keep the IP and OP phases are kept in kernel space to allow sharing of resources amongst applications.

3.3 CLF Queues

Though Figures 5 and 6 show constant durations for each pipeline stage, in reality execution time may vary. To accommodate this variation, FShm passes frames from one stage to the next using point-to-point queues. Each pipeline stage writes to one or more output queues and reads from a single input queue. Absent careful designing, the overhead of queuing and dequeuing data from could easily wipe-out all throughput gains.

In order to minimize the overhead of communicating among pipeline stages (in particular the locking overhead), FShm uses concurrent lock-free (CLF) queues [25]. Typically, these queues require no locking provided that the hardware supports an atomic memory exchange instruction. However, FShm uses only point-to-point queues with a single thread reading and writing each queue. In this special case, proper queue semantics can be guaranteed using the hardware's cache coherence mechanism rather than expensive high-level mutual exclusion primitives (see implementation notes below).

Notice that point-to-point only CLF queues in FShm also imply that there is no high-level contention in the queues. Just as there is a single read stage for the input NIC and a single write stage for the output NIC to avoid contention, each queue only has a single writer and a single reader.

At the low-level, there may be contention for the cache-lines that contain the queue entries. Ideally, if there are two active processors operating on a single queue, as in FShm, they will be processing queue entries separated by more than a single cache line, eliminating the overhead of cache coherence due to this contention. In practice, we demonstrate excellent performance even without cache-line separation.

More advanced mechanisms such as transactional memory architectures [15] may be able to mask the cache coherence time when two processors are sharing a single cache line. Furthermore, the transactional semantics allow low-overhead multipoint-to-multipoint queues, which can be advantageous.

The biggest advantage of CLF queues in FShm is seen when performing cross-domain messaging (e.g. process $\leftrightarrow$ process and process $\leftrightarrow$ kernel). In traditional message passing systems one would need to perform an expensive operation to synchronize the two processors or spin until it is safe to proceed. Neither technique is acceptable for the rates possible on modern networks. This does not happen with CLF queues.

3.3.1 CLF Queue Implementation Notes

The implementation of CLF queues creates a per-queue shared memory region containing an array and a shared control region that that manifest as a single device node to user-space applications. Utilizing device nodes provides a simple API that makes it easy to manage device specific calls and track misbehaving applications that forget to disconnect from a queue, without modifying kernel data structures. Recall from Section 2 that VFS and ioctl calls are prohibitively expensive. However, by FShm's design, these are only accessed on startup and shutdown.

Both the shared array and control regions are mapped read/write as there is a small amount of shared state. In addition there is a non-shared memory version that provides a process-local thread $\leftrightarrow$ thread communication mechanism suitable for user-space inter-thread communication.

The main implementation challenge for CLF queues is in ordering operations such that they only rely on memory coherence, not the memory consistency model (which varies across processors) or high-level mutual exclusion primitives. Figure 7 shows pseudo-code for the non-blocking put and get operations using the shared regions. This code works with all processor consistency models, as only coherence is necessary for mutual exclusion. To see this, recall that the queue is point-to-point so there is only a single thread calling put and a single thread calling get. Furthermore, put will only write to the shared region when the queue node is `NULL` and get will only write the node when the value is not `NULL`. Finally, since stores are atomic on all coherent memory systems, the getter and putter will never update the same node simultaneously.

Pipeline stages can dynamically switch to using blocking operations permitting process sleep during times of reduced load.

3.4 BSD Implementation Notes

Given a set of CLF queues, implementing a shared buffer region that is compatible with the existing BSD mbuf message passing system is straightforward. First a structure is defined that contains an mbuf as the first item, permitting us to typecast between the two as needed; we are breaking strict type rules so the programmer needs to be careful. Additionally, we store the kernel address of the structure and a pointer to the kernel address of its associated data buffer. The data buffer is functionally the same as the one the mbuf cluster allocator would create for a normal mbuf when allocating a buffer for a network interface. For convenience, our data buffers are set to 2048B easily preserving page alignment for the network interfaces. We track kernel addresses as they are constant for all applications; presently FreeBSD does not export an API to make kernel virtual addresses directly accessible by a user-space application.

3.5 Network Interfaces

Given the general purpose nature of the FShm architecture, the integration of the network interface is not a primary concern. A variety of different high-performance interconnection architectures are possible. We chose to slightly modify the stock drivers in FreeBSD for the Broadcom 5703 & 5704 and Intel 1000/Pro Ethernet network interfaces and attach the FShm architecture in the kernel. Alternative mechanisms could utilize user-space DMA or mapping the interface's transmit and receive descriptor rings to the application [9]. We chose to manage our interfaces in the kernel to demonstrate kernel$\leftrightarrow$process communication. Additionally, FreeBSD lacks an API to pin user-space threads which is needed for maximum performance with a user-space interface.

To manage the network interfaces we implemented two additional kernel modules to handle receiving (IP) and transmitting (OP). These modules each are capable of mapping one or more interfaces to one or more queues. A more detailed description of these modules can be found in Sections 4.3.1 & 4.3.2.

Presently the modules purely poll their inputs for work, although they could also be implemented with a hybrid interrupt and polling organization to conserve system resources and prevent livelock [26]. We chose not to implement this as we are demonstrating the limits of the system where every processor will be utilized at 100%, and therefore interrupts would never be activated.

3.6 Safety

Safety is a critical concern when sharing memory and passing memory references between processes. Any process at any time can misbehave and write invalid and or inconsistent values into the memory so that a correctly behaving process reads it and does the wrong thing. Examples of errors are writing an invalid address causing the reading process to examine the wrong memory and failing to ensure group-write operations are atomic in nature. In its evaluated form, FShm assumes that applications sharing the FShm memory region are not malicious and makes no serious effort to protect itself again misbehaving participants. This is a fair assumption as an application could be written as a single large multi-threaded application with user-space access to the network drivers where a fault would still terminate the application.

This does not imply that security was not a concern in the design of the system and several steps were taken to architect the system so that it could protect itself when necessary. First, access to the queues is performed though device entries and thus gains the underlying security model maintained by the host operating system. Second, the queues are in a separate region from the sbuf regions and the sbuf regions are split into an sbuf region and a data region. This division makes it possible for the kernel to share the sbuf header as read-only with user-space applications preventing the applications from corrupting the sbuf header itself (e.g. corrupting data pointers or the reference count field). This permits the application to only manipulate the frame data or the reference in the queue. Corruption of the frame data is not a concern for the kernel as it does not process it. However, corrupting the sbuf reference in the queue is a concern and can be dealt with by checking the address against the known bounds of the sbuf region. Premature processing of a frame in the kernel is also not a concern for the kernel itself as the worst case scenario is transmission of a garbage frame resulting in application failure but not system failure. Bounds checking can easily be performed in the available time for processing at any frame size as all the necessary information will be resident in the first level cache or registers.

Recovering from leaked buffers is the only difficult problem for the application to deal with and the only reasonable recovery mechanism is to restart the entire application. Selective restarting of application components may be possible for certain applications. However, the necessary bookkeeping adds significant overhead.

4 Evaluation

This section presents an evaluation of FShm.

4.1 Evaluation Criteria

To measure the effectiveness of FShm, three metrics are used: (1) frame latencies, both end-to-end and in-system, (2) wall-clock time available for frame processing, and (3) frame drop rates.

Each of these quantities is measured in frame generation, frame capture, and frame forwarding configurations, under a variety of frame sizes. The scenarios will be described in more detail below. Drop rates are measured by comparing how many frames should have been processed to the number of frames that were actually processed. Wall-clock time for actual work is measured by monitoring the drop rate as the duration of a dummy work loop is steadily increased. The time taken by the dummy work loop is measured using fine-grain timing instructions (`rdtsc` on x86 platforms) and the duration of the longest work loop that results in no increase in the drop-rate gives the wall-clock time for performing actual work on a frame.

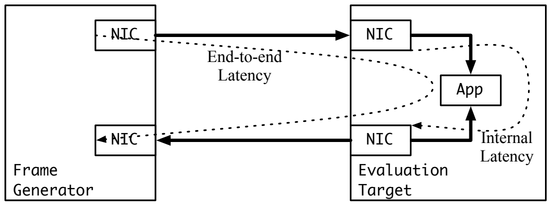

Figure 8: Frame Latencies

The two frame latency measurements are illustrated in Figure 8. For end-to-end latency, since the sender and receiver are on the same machine, the end-to-end frame latency is measured by inserting a 64-bit time-stamp (via the processor cycle-accurate TSC register on the x86 platform) when the frame is sent and comparing to the time the frame is received. Internal frame-latency is computed by time-stamping the frame when it is first seen in software on the processing machine and then comparing it to the time the frame leaves software control. For generation and capture of frames, we were unable to measure the end-to-end frame latency because the clocks (i.e., TSC register) of the frame generator and frame receiver are not synchronized.

4.2 Evaluation Platform

The network testing environment consisted of two AMD Opteron systems connected with either the on-board Broadcom 5704C (dual Gigabit Ethernet), off-board HP NC7771 (Broadcom 5703), or an Intel 82545 GM Network Interface Controller. The AMD Opteron systems were based on the Tyan Thunder K8SR (S2881) motherboard with dual Opteron 270 dual-core processors running at 2GHz. The relevant chipsets on the motherboard are the AMD-8131 (HyperTransport PCI-X Tunnel) and the AMD-8111 (HyperTransport I/O Hub).

4.3 Evaluation Modules

Each evaluation scenario uses a configuration with instances of four conceptual modules: receiver, transmitter, frame generator, and handler. Each module corresponds to a pipeline stage in Section 3 and runs as a thread on a dedicated processor, in the ideal case. The details of each module are described below.

4.3.1 Receiver

The kernel receiver module (KRx) corresponds to the IP stage and handles the task of moving frames from the input interfaces to the associated FShm queues. The module permits every NIC-to-queue mapping possible.

The receiver performs four tasks for the evaluation. First, it measures the average amount of free wall-clock time per frame by spinning in an idle loop. This gives a measure of the possible duration of early frame processing (e.g. selecting output queues). Second, by spinning the receiver smooths out the arrival rate of frames to make frame latency calculations more accurate by reducing bursting due to batched DMA. Third, the receiver measures the time it takes to enqueue the frame onto its output queue. From these two measures we can compute the amount of time spent polling the input interfaces. Finally, the receiver can timestamp frames so that the transmitter can measure in-system latency. There is also a specialized receiver module that computes the average frame round-trip time.

4.3.2 Transmitter

The kernel transmitter module (KTx) corresponds to the OP pipeline stage. It handles the task of moving frames from a set of incoming FShm queues to their associated network interfaces. Again, the module is designed to permit every queue to NIC mapping possible. The mapping of multiple queues to separate network interfaces is demonstrated in Section 4.6.

The transmitter performs two tasks for the evaluation. It optionally timestamps frames just before it enqueues them onto its transmit interface permitting round-trip latency measures as seen by the receiver module. In-system latency is also measured by comparing the current time to the receiver time-stamp when a frame is about to be enqueued to its output network interface.

4.3.3 Generator

Frame generation is performed by three different modules: KSend1, KSend, and USend. KSend1 is a dedicated kernel module that generates a single frame in a buffer and enqueues that same buffer for every frame transmitted. This avoids the latency in flushing the frame from the cache to main memory so the network card can DMA the frame without stalling. The remaining two modules (KSend and USend) operate by dequeuing an sbuf from the allocator queue, populating its data buffer with a frame, and enqueuing it onto the output queue for the transmitter module to output. KSend operates in kernel-space and USend operates in user-space. KSend and USend implement degenerate case APP stages.

4.3.4 Handler

The handler module is a simple APP stage whose primary task is to evaluate how much wall-clock work time is possible. We have implemented versions of the module to operate in both kernel-space (KHandle) and user-space (UHandle) permitting direct evaluation of the user-space scheduler. The effect of the user-space scheduler could be eliminated with process pinning, but FreeBSD does not support this. The handler is informed of the total per-frame service time allowing it to identify time taken by the scheduler in the absence of pinning.

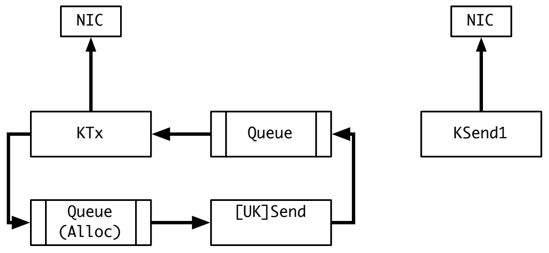

Figure 9: Setup: Generation

4.4 Generation

The generation scenario measures how well FShm allows kernel-space and user-space code to generate frames for transmission. For this set of experiments, one Opteron system (the evaluation target) running FShm is used as a frame generator. The NIC is connected to a dummy NIC that will sink frames at whatever rate they are generated.

The high-level arrangement of modules on the evaluation target is shown in Figure 9. On the left, we have the full generation setup using FShm. Here, the USend or KSend module pulls a free buffer (an sbuf in particular) from the allocation queue, fills the buffer and then enqueues the sbuf for the KTx module. The KTx module deallocates the buffer, sends the frame and adds the now free sbuf back to the allocation queue. Each queue is an FShm CLF queue that delivers frames in-order. KTx transmits all frames in-order. The right of the figure shows a simplified frame generation configuration that is used to test the maximum transmission rate supported by the hardware. Note that KSend1 communicates directly with the device driver and no other communication is needed.

Figure 10 depicts the results for the frame generation tests. The bar labeled Theoretic Max shows the maximum frame rate supported by Gigabit Ethernet for the given frame size. The remaining bars show the actual transmission rates for the various generation configurations. The bars labeled KSend1 are for the configuration to the right of Figure 9 and the remaining bars are for the configuration to the left. Recall that a U prefix signifies a user-space sender and the K prefix a kernel-space sender. The graph shows generation rates using a Broad-

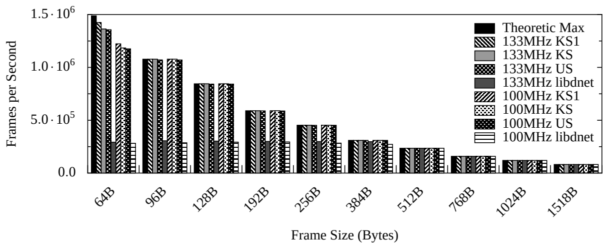

Figure 10: Generation Rates

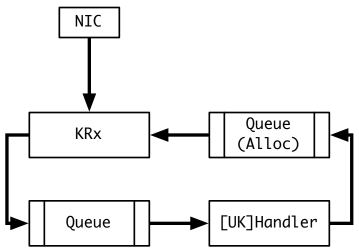

Figure 11: Setup: Capture

com 5703-based NIC on both the Opteron's 133 MHz PCI bus and the 100 MHz PCI bus. The bars labeled libdnet serve as a reference and show the frame transmission rates for a frame generator based on libdnet.

Notice that for all but the 64 byte frame size, FShm is able to generate frames at the theoretic maximum for Gigabit Ethernet. Clock skew accounts for the few bars that are slightly above theoretic maximum. For small frames, libdnet is not even close the theoretic maximum. For 64 byte frames, even $\mathrm{KSend1}$ only comes to within 95% of theoretic max showing that the hardware is the limiting factor. The FShm numbers lag the hardware limit by a bit, most likely due to cache coherence issues.

For all future experiments in which a frame generator is needed to saturate the link, an Opteron system using $\mathrm{KSend1}$ with the NIC on the 133 MHz bus is used to generate $64\mathrm{B}$ frames. The same Opteron and bus are used with KSend to generate the other frame sizes.

4.5 Capture

In this scenario a frame generator is connected to another Opteron system (the evaluation target) configured to capture frames. The configuration of FShm is shown in Figure 11. Again, each module runs in its own thread, and all queues are FShm queues. Here, the KRx module pulls a free sbuf off the allocation queue, fills the sbuf with data from the NIC and enqueues the sbuf for the Handler. The Handler consumes the frame and returns the now free sbuf to the allocation queue.

Results are shown in Figure 12 which breaks down

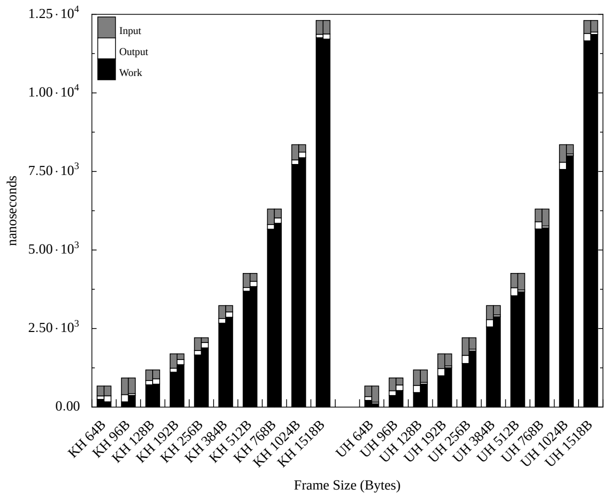

Figure 12: Capture Work

the amount of time available for work in-handler (work time), the time spent forwarding the frame to the next queue (output time), and the time taken to read the incoming frame (the input time). It is important to notice that the time spent handling the queues remains constant in both the kernel and user-space handler modules demonstrating FShm's scalability. Note that no frames are dropped at any frame size.

Compare this to BPF capturing the same generated frame stream, using stock FreeBSD drivers and the Broadcom 5703 NIC. Here, the drop rate exceeded 75% for small frames. Furthermore, only 99.95% of frames were captured, even for the largest frame sizes.

Observed latencies for the kernel-space handler are all bounded by twice the available service time. Since there are two pipeline stages in this degenerate scenario (IP and APP), this means that all frames are processed without drops. Although user latency could not be measured directly, the average latency can be bounded by experimentation and is within acceptable limits. This was evaluated by sending 10^9 frames (i.e., 64 GB of traffic and ensuring there were no drops. With eleven minutes of sustained traffic we can be confident that this throughput can be sustained indefinitely.

4.6 Forwarding

For the forwarding experiments, a frame generator sends frames to an Opteron system (the evaluation target). The frames forwarded by the evaluation target are then returned to the frame generator so that it may compute end-to-end latency, drop rates, and round-trip time.

Initially we evaluated the performance of a single thread forwarding frames from one NIC to another with-

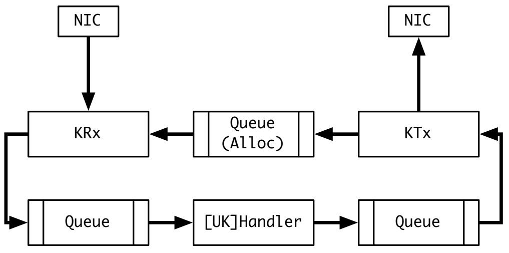

Figure 13: Setup: Forwarding

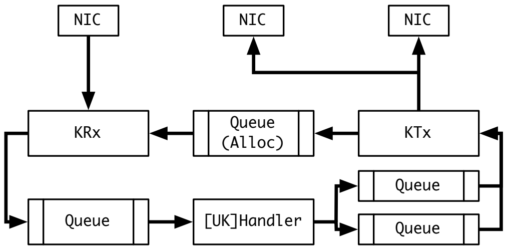

Figure 14: Setup: Forwarding 64-byte

out the FShm architecture and found a non-zero drop rate at every frame size, indicating that one thread cannot keep up with Gigabit Ethernet.

For the FShm test, the configuration of the evaluation target is shown in Figure 13. Forwarding is evaluated with exactly the same setup as the capture evaluation, only instead of the handler's output queue being the allocator, it is connected to a KTx module that outputs the frames on the appropriate NIC, in-order. The in-system latency, in the forwarding setup, must be bounded by three times the service rate for each frame as this setup has three pipeline stages. The external latency is shown in Table 4. This is measured by taking the average round trip time and dividing by two. Dividing the external la-

Frame	kernel-space		user-space	
Size	Time (ns)	# Frames	Time (ns)	# Frames
64B	N/A	N/A	N/A	N/A
96B	394373	213	528102	286
128B	484701	206	664873	282
192B	529621	157	650713	193
256B	716337	163	1212474	276
384B	1036671	161	1896708	295
512B	1353471	160	2178294	257
768B	1737235	138	2750763	219
1024B	2233380	134	3639886	219
1518B	3207739	131	5239745	214

Note: times for 64B frames were too small for accurate measurement.

Table 4: Forwarding System Latency

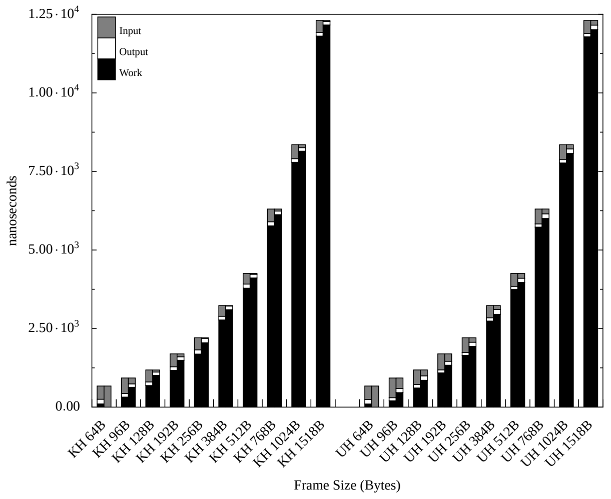

Figure 15: Forwarding Work

tency by the inter-frame arrival period, we can confirm that NIC queuing is causing the high latencies.

There is some difficulty evaluating forwarding of 64-byte frames. Section 4.4 shows that there is an hardware limitation in generating frames to a separate buffer at the full rate. This implies that to forward 64 byte frames at the maximum arrival rate, we need to use multiple output NICs. Therefore we used both output ports on the Opteron's on-board Broadcom 5704C NIC which internally times its DMAs to avoid interference on the bus. Also, a side effect of generating at the 95% transmission rate using the KSend1 module is that it is impossible to insert a timestamp into outbound frames to measure external latency. Furthermore, due to cache miss latencies in the KTx and KRx modules, it was not possible to insert timestamps to compute in-system latency. To compensate for the absence of direct measurement we used the sustained traffic technique from Section 4.5.

Figure 15 shows the amount of useful work that the handler can perform before the system begins to drop frames. The bar to the left of each bar-pair shows the amount of work (and overhead) that can be done in the IP stage (i.e., the receiver module) and the bar on the right quantifies the work in the APP (i.e., the handler module) stage. Notice that most of the time at large frame sizes is available for work. Note further that the total available work time is more that twice the actual inter-frame arrival period, verifying that FShm does indeed permit time-expansion through pipelining.

At smaller frame sizes, less work is possible. At the 64 byte frame size, FShm consumes almost all available time leaving no time for work. However, recall that FShm can expand work-time with additional processors, it can still allow real work to be performed. For example, the handler could forward frames to N additional handlers. This would give N times the per-frame processing time. The stream would then be reassembled at the output module. While existing commodity systems lack the processors to do this, we have demonstrated that the handler can split an incoming stream of 64 byte frames with no drops by testing the configuration in Figure 14. With the *upcoming* quad core dual processor systems, FShm will be more than able to do zero-drop in-order processing of 64 byte frames and still allow actual work to be accomplished, even though the original configuration in Figure 13 does not.

5 Related Work

As outlined in the prior sections, the FShm architecture builds upon a rich body of prior work by synthesizing ideas from the areas of operating system design [3, 13, 16, 22], message passing [2, 6, 28, 11, 5, 30, 12, 18], networking architecture [8, 10, 14, 19, 23, 31], and high performance computing[22, 25].

Of these works, the Synthesis Kernel[22] has been most influential. The Synthesis Kernel pervasively employed concurrent lock-free queues for passing messages between tasks, and is the inspiration for many of the ideas in FShm's communication mechanism. Specifically, FShm utilizes the single-producer/single-consumer array based queue for all message-passing.

Both FShm and NetTap[4], an existing system to permit high-speed user-space processing of packets, use CLF queues and shared memory to communicate between kernel and user-space contexts. NetTap focused on providing a specific API that was optimized for the single processor machines that were available on the commodity market at the time. They achieve satisfactory results. FShm concerns itself with providing a general purpose API designed to efficiently utilize multi-processor and multi-core commodity systems. Furthermore, given the time of publication, NetTap was evaluated on 100 Mbps Ethernet and may not fare well on modern networks.

Recently Intel has been working on extending the architectures of the Virtual Interface Architecture [12] and InfiniBand [17]. Their goal is to accelerate the processing of TCP/IP packets by dedicating a processor to act as a TCP/IP *onloading* engine [29]. This processor would preprocess packets and deliver them to applications through a set of queue structures with an interface optimized for TCP/IP. However, ETA does not pipeline frame processing and is not evaluated on small frame transmissions. Furthermore, FShm remains agnostic to the application whereas ETA is TCP/IP centric.

Finally, the Click Modular Router[19] is a network processing architecture with goals similar to those of the FShm. Click succeeds at providing a modular network processing infrastructure with general purpose user-space processing. The key to their success is that modules can be tested in user-space and then recompiled as a kernel module, thereby avoiding expensive communication between kernel-and user-space. By combining FShm and Click one can gain the benefits of both.

6 Conclusion

The flexibility and low cost of commodity hardware systems make them an appealing choice for network processing applications. However, as seen from Section 2, several bottlenecks in the software architecture make processing frames at Gigabit Ethernet rates difficult. Furthermore applications such as network intrusion detection (NID) must be able to process all frame sizes on saturated links. Prior to this work, this has not demonstrated on commodity hardware. Worse still, it appears as if prior work has eliminated almost all overhead for frame processing on single processor machines.

To surmount the barrier to wire-rate small frame processing, The FShm frame processing architecture, presented here, increases frame throughput by pipelining frame processing operations across multiple processors. To reduce the overhead of communicating between pipeline stages, FShm uses shared memory and concurrent lock-free queues to permit *user-space* applications to process all frames on a saturated Gigabit Ethernet link, at all frame sizes, including 64 byte frames. Furthermore, because FShm does not discriminate between user↔user, kernel↔kernel, and kernel↔user communication, it allows pipeline stages in both kernel and user-space. Finally, since the number of pipeline stages is limited only by processor resources, FShm can be used to split frame streams and expand per-frame processing time with additional processors. This makes FShm ideal for today's and the future's multi-core systems. Using FShm, we demonstrate that (1) FShm can indeed process all frame sizes at Gigabit Ethernet rates with no frame drops, (2) quantify that FShm permits useful per-frame work, and (3) that FShm can at least double the per-frame processing time through use of multiple processors.

7 Acknowledgments

This material is based in part upon work sponsored by ARO under Contract DAAD19-01-1-0484. The content does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. This work is also supported by the generosity of AMD Corporation and the National Science Foundation under grant CNS 0454404, "Wireless Internet Building Blocks for Research, Policy, and Education."

8 Availability

Project information and source will be made available at `http://systems.cs.colorado.edu/mediawiki/index.php/FShm`

References

[1] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems 10*, 1 (February 1992), 53–79.

[2] BERSHAD, B. N., ANDERSON, T. E., LAZOWSKA, E. D., AND LEVY, H. M. Lightweight remote procedure call. In *12th Symposium on Operating Systems Principles* (December 1989), pp. 102–113.

[3] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M., BECKER, D., EGGERS, S., AND CHAMBERS, C. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles* (Copper Mountain, Colorado, 1995), pp. 267–284.

[4] BLOTT, S., BRUSTOLONI, J., AND MARTIN, C. NetTap: An efficient and reliable PC-based platform for network programming, 1999.

[5] BRUSTOLONI, J. C. Interoperation of copy avoidance in network and file i/o. In *INFOCOM (2)* (1999), pp. 534–542.

[6] BRUSTOLONI, J. C., AND STEENKISTE, P. Effects of buffering semantics on I/O performance. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation* (New York, NY, USA, 1996), ACM Press, pp. 277–291.

[7] CHEN, T.-F., AND BAER, J.-L. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)* (New York, NY, 1992), vol. 27, ACM Press, pp. 51–61.

[8] DECASPER, D., DITTIA, Z., PARULKAR, G., AND PLATTNER, B. Router plugins: a software architecture for next-generation routers. *IEEE/ACM Trans. Netw. 8*, 1 (2000), 2–15.

[9] DERI, L. nCap: Wire-speed packet capture and transmission. In *Proceedings of E2EMON* (2005).

[10] DITTA, Z. D., AND ET AL. The APIC approach to high performance network interface design: Protected dma and other techniques.

[11] DRUSCHEL, P., AND PETERSON, L. L. Fbufs: a high-bandwidth cross-domain transfer facility. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1993), ACM Press, pp. 189–202.

[12] DUNNING, D., REGNIER, G., MCALPINE, G., SHUBERT, B., BERRY, F., MERRITT, A. M., GRONKE, E., AND DODD, C. The virtual interface architecture. *IEEE Micro 18*, 2 (1998), 66–76.

[13] ENGLER, D. R., KAASHOEK, M. F., AND J. O'TOOLE, J. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1995), ACM Press, pp. 251–266.

[14] HANDLEY, M., KOHLER, E., GHOSH, A., HODSON, O., AND RADOSLAVOV, P. Designing extensible ip routers software. In *NSDI '05: 2nd USENIX Symposium on Networked Systems Design and Implementation* (2005).

[15] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of theTwentiethAnnual International Symposium on Computer Architecture* (1993).

[16] HUTCHINSON, N. C., AND PETERSON, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (1991), 64–76.

[17] INFINIBAND TRADE ASSOCIATION. `http://www.infinibandta.org`.

[18] KHALIDI, Y. A., AND THADANI, M. N. An efficient zero-copy i/o framework for unix. Tech. rep., Mountain View, CA, USA, 1995.

[19] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. *ACM Transactions on Computer Systems 18*, 3 (2000), 263–297.

[20] libpcap. `http://www.tcpdump.org`.

[21] libpcap-mmap. `http://public.lanl.gov/cpw/`.

[22] MASSALIN, H., AND PU, C. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)* (1989), vol. 23, pp. 191–201.

[23] MCCANNE, S., AND JACOBSON, V. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter* (1993), pp. 259–270.

[24] METCALFE, R. M., AND BOGGS, D. R. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM 19*, 5 (July 1976), 395–404.

[25] MICHAEL, M. M., AND SCOTT, M. L. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. *Journal of Parallel and Distributed Computing 51*, 1 (1998), 1–26.

[26] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems 15*, 3 (1997), 217–252.

[27] OLSSON, R. pktgen the linux packet generator. In *Proc. linuxsymposium 2005* (2005), vol. 2, pp. 11–24.

[28] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Io-lite: a unified i/o buffering and caching system. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation* (Berkeley, CA, USA, 1999), USENIX Association, pp. 15–28.

[29] REGNIER, G., MAKINENI, S., ILLIKKAL, R., IYER, R., MINTURN, D., HUGGAHALLI, R., NEWELL, D., CLINE, L., AND FOONG, A. Tcp onloading for data center servers. *IEEE Computer 37*, 11 (2004), 48–58.

[30] VON EICKEN, T., BASU, A., BUCH, V., AND VOGELS, W. U-net: a user-level network interface for parallel and distributed computing (includes url). In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1995), ACM Press, pp. 40–53.

[31] WELSH, M., CULLER, D., AND BREWER, E. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), ACM Press, pp. 230–243.