Frame Shared Memory:
Line-Rate Networking on Commodity Hardware

ABSTRACT
Network processors provide an economical programmable platform to handle the high throughput and frame rates of modern and next-generation communication systems. However, these platforms have exchanged general-purpose capabilities for performance.

This paper presents an alternative; a software framework for commodity general-purpose architectures capable of high-rate and throughput sequential frame processing compatible with high-level languages and general-purpose operating systems. A cache-optimized concurrent lock free queue provides the low-overhead core-to-core communication necessary to sustain sequential frame processing rates beyond the 1.41 million frames per second realized on gigabit Ethernet while permitting per-frame processing time expansion with pipeline parallelism.

1. INTRODUCTION
An ideal frame/packet processing platform would permit an application to sequentially process frames at line-rate and expand the available per-frame processing time with parallelism. The ability to process frames sequentially is important as an increasing number of systems use state that is updated as each frame is processed. Example systems include network intrusion detection systems and any requiring upper layer packet processing [36]. Time expansion, the ability to trade latency for processing time while maintaining throughput, is similarly important as modern networks can sustain traffic with frame interarrival periods on the order of 10’s to 100’s of nanoseconds.

Simultaneously supporting both time expansion and sequential processing, in the limit, requires a pipeline-parallel platform. Note that a connection- or data-parallel environment is insufficient for scenarios requiring stateful inspection of every frame in sequence. The difficulty is that pipeline parallelism’s design requires that every stage consumes no more than the minimum arrival time for every given frame size to accomplish work and stage-to-stage communication.

In the past, designers turned to fully custom processing engines to satisfy these requirements. Recently, however, designers have shifted to special purpose programmable platforms (e.g., network processors). This shift has dramatically reduced both the cost and time needed to develop a system. Unfortunately, these special purpose platforms typically expose low level platform-specific implementation details that developers must properly manage to achieve full performance. For example, network processors typically scale the memory wall [38] and achieve the performance constraints imposed by modern networks by exposing their architecture to the developers. Exposed elements have included processor interconnections, explicitly managed memory hierarchies, and lightweight threading. Thus, developers are forced to forgo the niceties of general purpose languages and traditional operating-system support. These details complicate application development and harm portability by coupling the software to a specific platform.

Recently, researchers have found that the performance of caches (as found on general purpose processors) outperform specialized memories and exposed memory hierarchies [20, 29]. Furthermore, general purpose systems have begun to borrow from the designs of special purpose systems, while still supporting standard languages and operating systems. For example, general purpose processor designers now focus on multi- and many-core designs and improving interconnection performance [1,4,9,35]. This focus has given rise to systems incorporating high performance crossbars, switched fabrics, and heterogeneous computing environments.

This paper presents the Frame-Shared-Memory (FShm) software architecture. FShm exploits modern multicore general purpose commodity platforms to deliver high-rate frame processing comparable to those of special purpose platforms. FShm uses each core on a multicore system to implement individual stages in a processing engine. To minimize stage-to-stage communication delays, FShm uses FastForward, a new cache-optimized core-to-core single-producer/single-consumer concurrent lock-free queue [2, 3]. Because FastForward relies only on shared-memory and cache-coherence, it can link processing stages that are inside an operating system’s kernel, user-space applications, or any other device connected to the memory interconnect. Notice that since any memory-connected device can be a processing stage, application designers can substitute a hardware stage for a software stage and vice-versa without altering any of the other processing stages (e.g., on-die special purpose processors [1], network processors [39], or FPGAs [8]).

This paper concludes that general purpose systems are a viable alternative to network processors and may continue to be so as general purpose systems evolve. FShm on a 2.0 GHz AMD Opteron with PCI-X network interface cards is capable of supporting frame capture and generation at 1.41 million frames per second, and nic-to-nic forwarding at 1.37 million frames per second, sufficient for line-rate processing of 74B and larger frames.
on Gigabit Ethernet. Further we show that the architecture can support substantially higher frame rates as defined by the input medium and desired processing time.

The remainder of this paper is organized as follows. Section 2 discusses the criteria considered when designing FShm. Section 3 discusses the operating system and hardware constraints that were addressed in the FShm design. Section 4 surveys previous work. Section 5 discusses the FShm’s software architecture in detail. Section 6 presents the evaluation results. We conclude in Section 7.

2. DESIGN CRITERIA
Frame Shared Memory (FShm) is designed to meet a collection of constraints that balance the need for high-performance in research and deployment of frame processing systems against the flexibility offered by general-purpose systems. The first two constraints establish the baseline performance for research and deployment of frame processing applications on current- and next-generation networks.

1. Frame capture, generation, and forwarding at rates close to the communication system’s limits.
2. Real per-frame work must be possible for all frames at all legal sizes.

FShm’s design is further constrained to ensure a general-purpose framework. These constraints focus on several properties common to many network applications and development environments.

3. A general purpose mechanism must be available to trade-off latency for increased per-frame processing time. Modern and future processors are unlikely to individually have the processing time to support the high-rates available on next generation communication systems, such as 10 gigabit Ethernet where a new frame may arrive every 67 ns.
4. In- or out-of-order frame processing must be possible without noticeable performance impact. In-order processing is desirable for network intrusion detection systems, while out-of-order processing is desirable to implement quality-of-service in routers.
5. Composing frame processing stages into an application must be straightforward [19]. Ideally, software and hardware stages should be composable.
6. High-level languages and operating system environments must be supported to enable rapid application development and deployment.

3. SYSTEM CONSTRAINTS
The criteria in Section 2 are challenging for any system and require managing system specific elements. On general-purpose systems there are two primary concerns, the host operating system and the underlying hardware. Both concerns must be managed in the least invasive way possible to ensure the system remains general-purpose, is not micro-managed into a special-purpose system, but still meets performance requirements.

The primary performance constraint is the potentially high frame-rates imposed by the network itself. These rates can easily result in livelock [28] or dropped frames if a system’s general-purpose mechanisms exceed the interframe arrival period. On Gigabit Ethernet there maybe up to 1,488,095 frames per second, a new frame every 672 ns. On 10 Gigabit Ethernet, the situation is 10× worse with only 67 ns between frames. As we will see, both the software and hardware layers of general-purpose systems must be managed to manage these frame rates.

3.1 Operating System Constraints
General-purpose operating systems focus on efficiently providing a wide range of services to a mixed set of users. This design trade-off has, until now, relegated high-rate frame processing applications to special-purpose systems. The key will be to subvert only those performance-critical features without affecting other system tasks.

Thread pinning (i.e., dedicating a processor to a particular thread) for indefinite periods is a necessary feature for high-rate frame processing. Without thread pinning, scheduler activity can cause enough processing-time jitter to force frame drops at high frame rates; no amount of queuing or slack can accommodate these disruptions. Fortunately, most modern operating systems provide this feature.

Instead, the major operating system constraint results from safety guarantees for general-purpose interprocess communication. Systems traditionally enforce safety via copy semantics on all communication. These copies are a burden when frequently communicating parties are closely coupled. Below we discuss the impact of various features used to enforce safety: system calls for communication, locks for safe data exchanges, and providing copy semantics.

3.1.1 System Calls
System calls provide the traditional mechanism to bypass an application’s protected memory space and interact with both operating service and other processes. For most interactions, the overhead of system calls is irrelevant as their cost is negligible compared to the interaction itself. However, this is not the case when handling the frame rates on modern communication systems. Table 1 quantifies the costs of performing system calls on a dual processor dual core 2.0GHz AMD Opteron 270, running FreeBSD 5.5. In a process without the threading library (VCPU=0) a null system call will take on average 171 ns or 25.5% of the arrival period of a 64B frame on Gigabit Ethernet. If threading is activated (VCPU=1 or 2), times are much longer due to system
<table>
<thead>
<tr>
<th># VCPU</th>
<th># Thr.</th>
<th>System Calls (ns)</th>
<th>ioctl (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>171</td>
<td>538</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>851</td>
<td>1826</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1334</td>
<td>1674</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>911</td>
<td>1284</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>912</td>
<td>1223</td>
</tr>
</tbody>
</table>

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

Table 1: Opteron System-call and ioctl costs.

<table>
<thead>
<tr>
<th>Lib</th>
<th>Lock (ns)</th>
<th>Unlock (ns)</th>
<th>Lock &amp; Unlock (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread</td>
<td>75</td>
<td>78</td>
<td>159</td>
</tr>
<tr>
<td>kernel</td>
<td>28</td>
<td>19</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 2: Mutex costs for an AMD Opteron

3.1.2 Mutual Exclusion

Once a protected memory space has been entered in a controlled fashion, some form of mutual exclusion, often a mutex, is typically used for safe access to a shared data structure. While such mechanisms ensure safety, they serialize any threads attempting to access the protected data, coupling their execution. Additionally, the mutex overhead is high, even in the optimal situation where any necessary code and data are cache-resident and there is no contention. Table 2 quantifies the costs of mutex operations in kernel- and user-space for the AMD Opteron described earlier. The minimum overhead of sharing a variable between two threads is double the cost of a single lock/unlock pair since both the reader and writer must lock the data structure when performing their operations. Assuming this unrealistically optimal scenario, the expected cost of a lock/unlock pair is on the order of 318 ns for user→kernel/user or 54 ns for kernel→kernel/user communication, a substantial portion of the per frame time budget on Gigabit Ethernet or higher-rate networks.

3.1.3 Copy Semantics

Traditionally copy semantics have been used to ensure safe operation of communicating parties as both threads will have separate copies of the data after a communication. In some cases the copy overhead can be minimized by using a variety of zero-copy techniques [18] and/or by the sender relinquishing access to the transmitted data [11]. However, in all cases the overhead is non-negligible as the hardware memory management unit needs to be updated, an action requiring privileged hardware access.

Because these 3 constraints are difficult to manage, FSshm and other recent prior work [24, 33] subvert some of these interfaces to improve performance. In particular, shared memory and concurrent lock-free data structures [27] are used to accomplish the necessarily low-overhead interprocess communication. Neither has an effect on the rest of the system’s operation and thus meet the needs of our desired criteria.

3.2 Hardware Constraints

Hardware design provides interesting bottlenecks that are more difficult to control than the OS bottlenecks described above. The overall system I/O architecture and memory subsystem play the most significant role. The processor and memory interconnects and local-bus adapter cause most issues, and thus discussed below. However, since it is difficult for software techniques to mitigate these bottlenecks, they ultimately limit the performance of any system.

3.2.1 Processor Interconnects

Consider the potential bottlenecks in Figure 1 and Figure 2. In the host bus architecture, the primary potential bottleneck is the shared communication bus between the two processors. This shared bus may be problematic since access to memory, access to snoop results from other processors, and access to the PCI bus all contend for the same interface.

The switched system architecture shown in Figure 2 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 typically not a full crossbar.

3.2.2 The Local-Bus Adapter

Even if the system interconnect is not a problem, the internal organization of the local-bus adapter 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 another PCI bus. Other issues in the design, such as interrupt scheduling and command latency, can also have significant impact on total achievable throughput to NICs on the PCI bus.

Table 3 compares the number of minimum size Ethernet frames that can be generated on a Pentium III system and Pentium 4 Xeon system with Linux pktgen [30]. 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. 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.

Fortunately, despite these constraints, we will see that with FShm modern commodity systems can still deliver Gigabit Ethernet performance, with strong evidence that higher frame rates are possible (see Section 6). Further, the situation should improve as interconnect performance improves in response to the demands of traditional applications.

### Table 3: Pentium Throughput Comparison.

<table>
<thead>
<tr>
<th>System</th>
<th>NIC</th>
<th>PCI Bus</th>
<th>Frame Size</th>
<th>Max frames/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 GHz PIII</td>
<td>Intel 8245 GM</td>
<td>64-bit</td>
<td>66 MHz</td>
<td>64B 811 kfps</td>
</tr>
<tr>
<td>2.66 GHz P4 Xeon</td>
<td>Intel 8245 GM</td>
<td>64-bit</td>
<td>133 MHz</td>
<td>64B 723 kfps</td>
</tr>
</tbody>
</table>

4. RELATED WORK

Previous efforts focused on maximizing hardware resource utilization, typically to improve the cost effectiveness of the computing platform. FShm, in contrast, focuses on maximizing the performance of a specific class of applications, communications systems, by monopolizing the necessary resources without impacting general purpose functionality.

FShm achieves its performance while maintaining a general purpose computing platform by synthesizing a new general structure for high-rate computation from subcomponents of a rich body of work. Areas of influence include operating system design [13, 16, 25, 32], general purpose interprocess communication and message passing [5, 7, 11, 12, 18, 31], general purpose networking on commodity hardware [15, 19, 26, 37], general purpose high-performance networking [6, 10, 12, 17, 33, 34], and concurrent lock-free data structures [2, 3, 21, 22, 27].

From this body of prior work Synthesis, Lamport’s CLF Queue, FastForward, NetTap, nCap, ETA, and Click are most closely related to portions of FShm, and thus FShm shows the same trends on these systems.

Another related frame processing system is nCap [10]. Frame buffers are efficiently shared with a user-space application by mapping the transmit and receive ring buffers of the network card into a shared memory region. While this technique is the most lightweight of any, it fails to meet criteria 3 and 5 (Section 2). FShm instead concerns itself with providing a general purpose API designed to efficiently utilize multi-processor and multicore homogeneous or heterogeneous commodity systems.

The Synthesis Kernel [25] achieved dramatic results in the area of high-rate processing on one- and two-processor machines. The key to success was the hyper-optimization of the system with CLF queues, control theory based thread scheduling, auto-generated routines to minimize overhead (e.g., context swap routines), and online assembly recompilation and autotuning. The Synthesis lesson is that dramatic performance improvements can be had by maximizing the utilization of the system through micro-management of all system aspects. Unfortunately, the entire operating system was hand-coded in assembly on a custom machine and thus does not qualify as a commodity general purpose system. However, FShm does borrow the notion of fine-grain pipeline parallelism from Synthesis.

Lamport described a general purpose single-producer/single-consumer CLF queue and proved its correctness under sequential consistency [14, 22], and is used in Synthesis. However, this queue has two undesirable properties. First, communicating processors are not decoupled on cache-based systems, incurring significant overhead due to cacheline thrashing. Second, it is unlikely to work on future systems with very relaxed consistency models. Fortunately, FastForward [2, 3] is a CLF queue that addresses both issues: FastForward’s role in FShm is discussed in Section 5.2.

Both FShm and NetTap [6], an existing system to permit high-rate user-space processing of packets, use CLF queues and shared memory to communicate between kernel and user-space contexts. However, NetTap focused on providing a specific API that while correct on multi-processors, employed spin-locks for correctness, and was optimized for the single processor machines that were available on the commodity market at the time. Unfortunately, the API was designed for a single application stage and thus does not meet criteria 3 and 5 (Section 2). FShm instead concerns itself with providing a general purpose API designed to efficiently utilize multi-processor and multicore homogeneous or heterogeneous commodity systems.

Recently Intel has been working on ETA [34], an extension of the Virtual Interface Architecture [12] and InfiniBand [17]. The goal is to accelerate the processing of TCP/IP packets by dedicating a processor to act as a TCP/IP onloading engine [33]. This processor would preprocess packets and deliver them to applica-
tions through a set of queue structures with an interface optimized for TCP/IP. ETA extended Pentium 4 Xeon processors with two new interesting facilities that would be of general interest, first they implemented a light-weight threading systems as well as a cache-push instruction. FShm would benefit greatly from the cache-push and application developers would benefit from the light-weight threading. However, ETA fails to meet criteria 3 and 5 as it is very TCP/IP centric and does not facilitate parallelism beyond the TCP processing. Further, ETA was not evaluated on small frames.

Finally, the Click Modular Router [19] is a network processing architecture with goals similar to those of the FShm. However, to maintain portability, they suffer in user-space performance as they rely on available libpcap [23] libraries for frame input and output. Ideally Click modules will be implemented using the FShm architecture to provide low-overhead inter-process communication and thus avoiding the need to push Click modules into the kernel for performance.

5. FSHM DESIGN

This section presents the design of the Frame Shared Memory architecture. FShm leverages multicore general purpose systems and the FastForward cache-aware concurrent lock-free queue by Anonymized [2, 3] to realize a high-rate software system capable of scaling to millions of frames per second on commodity hardware.

Meeting the performance and functionality criteria enumerated in Section 2 requires that FShm not only operate at line-rate while doing useful work, but that it is possible to expand per-frame processing time, process frames in any desired order, be modular, and interoperate with languages and service available on general purpose systems. To maximize available work time and compatibility with existing interfaces, FShm only has the minimal communication support and organizational structure necessary. Specifically the framework consists of an implementation of the FastForward queue, a networking tuned pipeline balancing routine, and a BSD mbuf compatible shared buffer wrapper. The application developer is therefore freed to construct any desired pipeline or frame processing graph.

The design is discussed below in terms of its components, beginning with a discussion of a basic processing pipeline implemented using FShm and its extension to more general designs. Section 5.2 discusses how FShm utilizes the FastForward to achieve it's performance. Section 5.3 discusses the necessary shared frame buffer extension. Section 5.4 concludes the design section by discussing the safety issues implicit in eliminating interprocess copy semantics. We do not discuss implementation details beyond stating that they are slimmed down FreeBSD drivers with their transmit and receive descriptor management routines exposed.

5.1 The FShm Pipeline

The key to FShm’s ability to sequentially process small frames at line-rate comes from its pipeline parallel design that allows for expanded per-sequential-frame processing time. Note that the communication mechanism as described in Section 5.2 allows for more general parallel processing organizations as well.

Figure 3 depicts an overview of the three basic stages in a FShm processing pipeline. The basic stages are Input (IP), Application (APP), and Output (OP). There are two degenerate cases where there is no IP or OP stage corresponding to frame generation and frame capture systems respectively. The pipeline timing diagram is realized by binding each stage to a processor and connecting them with the provided low-overhead stage-to-stage communication mechanism (not depicted). From this figure we can see that in the ideal case where each stage takes the same amount of time, it is possible to triple the throughput of the system by simply overlapping sequential data on each stage.

Typically, this result is used triple the available per-frame processing time as the rate is externally imposed by the network. This can be achieved by permitting each stage in the pipeline to consume the full inter-frame arrival time (minus any communication overhead). Extending this pipeline to four or more processors permits the stages to be split into sub stages, each of which could consume the full inter-frame arrival time but permit additional processing time. Therefore it is possible to trade off per-frame latency for more processing time without impacting the throughput.

Extending the pipeline to data-parallel or other parallel organizations is straightforward and simply requires connecting stages in other organizations. Figure 4 depicts a three stage pipeline with a 2-way data-parallel application. The difference between Figures 3 & 4 is that the APP phase has been duplicated and the IP and OP stages access each of the stages in a synchronized round-robin fashion. This organization permits each APP phase to consume twice the interframe arrival rate without affecting the system’s throughput or frame rate. The pipeline timing diagram confirms this.
The obvious caveat of a data-parallel organization is that maintaining shared state between frames may be impossible at high-frame rates.

The two main challenges in realizing an FSshm design are (1) balancing stage execution time to guarantee maximal throughput and (2) minimizing stage-to-stage communication delay. Since, in reality, the IP, OP, and APP stages may not have the same duration, the pipeline will be unbalanced. However, even in the case of slight imbalance in stage timing, it is still possible to increase throughput albeit sub-optimally. Minimizing stage-to-stage delay is discussed below.

5.2 Stage-to-Stage Communication

Managing stage-to-stage communication overheads is critical in high-rate applications as any overhead becomes proportionately more expensive as the rate increases and interframe arrival time decreases. To minimize communication overhead, FSshm uses the FastForward CLF queue [2,3]. FastForward provides very low-overhead operations with sub-memory access latencies. Furthermore, stage-to-stage communication is the same regardless of whether the communication is intra- or interprocess. Finally, the queue interface is abstracted so that it is straightforward to replace the communication queues without needing to recompile the stages (criteria 5 & 6).

5.2.1 FastForward Tuning

FastForward achieves its performance by fully decoupling its operations at the data structure level by ensuring that in the steady state it is not necessary to share cachelines. In contrast Lamport’s queue [22] compares head and tail indices for every operation forcing cache line transfers for every operation in the steady state. Further, unlike other CLF queues, FastForward relies only on coherence for correct operation and is thus correct on strong to weakly ordered memory consistency models that will be found on future large multicore commodity systems.

While these features make FastForward appealing there are two primary concerns in using FastForward. First, there is a potential engineering tradeoff. To see why this is not an issue with FSshm, a quick review of the FastForward algorithm is necessary. Figure 5 contains the pseudo-code fragment of the non-blocking put routine. The performance gain over Lamport’s queue [22] is realized by preventing both the putter and the getter from competing for exclusive cache access to (1) the head and tail pointer, and (2) the data buffer itself (queue in the figure). The head-tail decoupling is done by using the data array itself to maintain full/empty status. Thus, the putter need only reference head, and the getter need only reference tail. This works transparently and is of no special concern in FSshm.

To avoid conflict on the data buffer (queue), FastForward requires (for performance) that enough elements are in the queue so that the producer (putter) and consumer (getter) are writing at least one cacheline apart, thus decoupling operation as described in [3]. This has the immediate effect of increasing latency but dramatically increasing performance. Fortunately, this buffering requirement is also needed in FSshm to account for interstage jitter and different frame sizes flowing through the pipeline. Therefore for the purposes of FSshm, this latency/throughput tradeoff is not an issue, provided that this producer-consumer separation can be maintained. However, to maintain a minimum number of elements in the queue the pipeline stages must be time-balanced, otherwise a faster stage will empty the queue and cause performance degradation. Automatic pipeline balancing, described below, is thus a key innovation for FSshm.

5.2.2 Automatic Pipeline Balancing

Unbalanced pipelines are present in all but the most micro-managed pipelines and must be handled efficiently. Typically FSshm stages will be unbalanced relative to each other for any given frame size. In most situations polling one’s input or output queue will automatically introduce the necessary stalls to keep the pipeline implicitly balanced (on average) by stalling on full and empty queues. However, FastForward’s performance depends on a minimum amount of queuing between a queue’s producer and consumer. There are two techniques that can be used to keep stages balanced. The first is to pre-compute the maximum stage time for each frame size and pad all stages by spinning on a cycle ac-
curate time stamp counter (e.g., TSC in x86). The second method is to periodically poll the index of the producer (head) and calculate the distance from tail. If the distance is below a low watermark (i.e., dangerously close to the FastForward threshold), spin until it has increased to the desired average distance. This second technique was used in the evaluation modules. Beware, that this technique incurs cache misses and therefore must be used as infrequently as possible.

5.3 The Shared Frame Buffer Wrapper
To maintain compatibility with the host operating system (FreeBSD in this implementation), simplify interfacing with existing network drivers, and permit the frames to be routed to any system component, FShm wraps each frame buffer with a BSD mbuf compatible header adding a level of indirection to access the payload. Additionally, FShm tracks these frames by their kernel address as FreeBSD does not permit the mapping of kernel memory addresses to user-space processes. In practice we found the overhead for both to be acceptable. For increased performance it may be advantageous to add the BSD mbuf compatible header in a lazy fashion to eliminate the level of indirection when not necessary. Notice that the indirection is a tool used to interface with the host OS and is not a limitation of the FShm architecture itself.

5.4 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.

The design of FShm incorporates several levels of protection. First, acquiring access to the queues is through device entries and thus gains the underlying security model maintained by the host operating system. Second, the queues are in a separate shared-memory 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 FShm 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.

6. RESULTS
To evaluate FShm we evaluate its performance on Gigabit Ethernet and then evaluate the architecture against higher frame rates.

First, we determine the maximum frame generation, capture, and forwarding rates possible without dropping any frames during a 1,000,000 test frame sequence on Gigabit Ethernet. Each scenario has a 3 stage pipeline with one or more handler stages that simulate an application stage work by spinning on the x86 time stamp counter (TSC) to measure wall clock work time. The TSC measures elapsed cycles since boot in a 64-bit register and is accurate to within a few cycles on our systems. Additionally there maybe an input and/or an output stage which manage the network interfaces. The input stage also measures the time available for work. Figure 6 depicts the evaluated forwarding pipeline; generation and capture scenarios are obvious variants of this simple pipeline and described below.

Second, we extrapolate the performance on higher-rate networks by using the values measured in the Gigabit Ethernet evaluation. Specifically we found that the queue time in the handler stages was sufficiently constant to connect three handler stages in a loop and doing a specified amount of work per stage.

6.1 Evaluation Platform
The network testing environment consisted of two AMD Opteron systems based on the Tyan Thunder K8SR (S2881) motherboard with dual Opteron 270 dual-core processors running at 2GHz. The relevant chipsets on this motherboard are the AMD-8131 (HyperTransport PCI-X Tunnel) and the AMD-8111 (HyperTransport I/O Hub). For networking, off-board HP NC7771 (Broadcom 5703) and Intel 82545 GM network interface cards were used in the 133 MHz and 100 MHz PCI-X slots that were on two separate PCI-X busses. The Intel card was used for receiving as it showed an advantage in receiving. Note that this system has a switched interconnect architecture such as the one in Figure 2.
6.2 Frame 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.

First we established the maximum rate possible by the NIC and driver combination by generating a frame on a single buffer and inserting the address into all of the NIC’s transmit descriptors for 1,000,000 frames. With 64B frames we were able to sustain approximately 1.41 million frames per second (mfps). With 74B frames or larger we were able to sustain theoretic max.

Second we created a test setup that had a stage which would read a frame from the allocator queue, copy a frame of the desired size into it, spin for a specified amount of time, and forward it to the output stage. The handler stage would read the frame, spin for a specified amount of time, and forward it to the output stage. The output stage reads frames from the input queue, and inserts them onto the NICs transmit descriptor ring. The output stage also removes the completed descriptors and forward the frames back into the allocator queue. With this setup we are able to generate frames at approximately 1.39 mfps with 64B frames and theoretic max at 74B frames. Figure 7 summarizes the available per-frame work times and I/O overhead for both the generation stage (left bar) and the handler stage (right bar).

6.3 Frame Capture

For the frame capture evaluation the performance of the frame capture configuration was not exceeded by the system’s generation capabilities even at 1.41 mfps. The platform is similar to the generation only the input stage is utilized along with two handler stage. The results shown in Figure 8 show that the total available per-frame processing time in the input stage (left bar) and handler stages (right two bars) is significantly expanded through the use of pipeline parallelism and can be extended with additional processors. Notice that the queue time is constant and equal for both handler stages at all frame sizes, same for the input stage.

6.4 Frame Forwarding

For the frame forwarding experiments the input, handler, and output stages used in the capture and generation experiments are linked together into a three stage pipeline (see Figure 6). Figure 9 shows full results, the left bar is for work in the input stage and the right bar for the handler stage. In this experiment, FShm successfully forwarded at theoretic max for 74B frames and higher. 64B frames were successfully forwarded at 1.33 mfps. We suspect the slight performance drop is due to the frame not being cache resident while being streamed to the output triggering extra transfers across the HyperTransport to distant memory modules (the dual-processor/dual-core AMD Opterons have a Non Uniform Memory Access architecture). However, the queue overhead for the handler stages remained negligible for all frame sizes. Further the input handler had available time for work suggesting it could manually pull the frames into cache before forwarding.

6.5 Extension to Higher Frame Rates

Figure 10 predicts the available work time at higher frame rates. Thus, the figure shows the future potential of the FShm system. Necessarily, these results as-
sume that input and output NICs can be constructed in such a way as to support the desired frame rates. From the figure, notice that FShm is capable of sustaining 10 mfps on the evaluation hardware, with 5 mfps as the practical cut-off for real work on our aging hardware platform². These rates are sufficient for realistic line-rate processing of 230B frames on 10 Gigabit Ethernet. As processor, memory, and interconnect speeds improve, FShm will scale to faster rates.

7. CONCLUSION

This paper presented the Frame Shared Memory architecture, a software-only framework for high-rate network processing on commodity multicore systems. With FShm, general purpose commodity multi-processors systems should be considered first-class citizens in advanced networking research and deployment. Evaluation demonstrated that such systems can be used to process frames at rates sufficient for line-rate processing of gigabit Ethernet and beyond. This performance is sufficient for deploying products on many current generation networks and researching next generation protocols. Furthermore, with the convergence between general- and special-purpose systems promising improved memory subsystems, interconnects, and processor counts, general-purpose systems will continue to improve as networks become faster.

While development on a general-purpose platform may not completely obviate the need for architectural tuning of algorithms (as necessary on network processors), developers now have the option of using general purpose systems features including operating system services, choice of operating system, high-level languages, and the ability to replace application components with software or hardware components without redesigning the whole platform or application. Furthermore, a framework like FShm minimizes architecture-specific tuning by hiding much of the performance tuning for optimal use of the multiple processors and high-performance interconnects found on current generation systems. Thus, we conclude that for many applications, using FShm, despite the slight performance penalty over special purpose systems, is a solid option when developing current generation line-rate processing systems and researching next generation systems.

8. REFERENCES


Operating systems principles, 1993.


