Frame Shared Memory:
Line-Rate Networking on Commodity Hardware

John Giacomoni
University of Colorado
Boulder, CO, USA
jgiacomo@colorado.edu

John K. Bennett
University of Colorado
Boulder, CO, USA
jkb@colorado.edu

Antonio Carzaniga
University of Lugano
Lugano, Switzerland
antonio.carzaniga@unisi.ch

Douglas C. Sicker
University of Colorado
Boulder, CO, USA
sicker@colorado.edu

Manish Vachharajani
University of Colorado
Boulder, CO, USA
manishv@colorado.edu

Alexander L. Wolf
Imperial College London
London, United Kingdom
a.wolf@imperial.ac.uk

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 network processor (Soft-NP) framework using commodity general-purpose platforms 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 necessary low-overhead core-to-core communication for sustained sequential frame processing beyond the realized 1.41 million frames per second (Gigabit Ethernet) while permitting per-frame processing time expansion with pipeline parallelism.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

software network processor, parallel programming, multi-core, multiprocessors

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 [35]. 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 excessive low level platform-specific implementation details that developers must properly manage to achieve full performance. For example, network processors typically scale the memory wall [37] 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 [19, 28]. 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 intercon-

nection performance [1,2,7,34]. 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 network processor (Soft-NP) architecture. FShm exploits modern multicore general purpose commodity platforms to deliver high-rate frame processing comparable to those of special purpose platforms with reduced developer effort. FShm uses each core on a multi-core system to implement individual stages in a processing engine. To minimize stage-to-stage communication delays, FShm uses FastForward, a new cache-optimized concurrent lock-free queue [13]. 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 [38], or FPGAs [6]).

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 1.36 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, generatation, 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 exchange 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 [18]. 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 micromanaged 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 [27] 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 $10\times$ 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 is 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. Interrupts on processors running FShm nodes are disabled for similar reasons. 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 safety guarantees are a burden when frequently communicating parties are closely coupled. Below we discuss the impact of various features used to enforce safety including system calls for communication, locks for safe data exchanges, and providing copy semantics concluding that they must be avoided.

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. Experimentally we found that system calls on a dual-processor dual-core 2.0 GHz AMD Opteron 270, running FreeBSD 5.5 took on average 170 ns or 25.5% of the arrival period of a 64B frame on Gigabit Ethernet. Passing frames with an ioctl, or other operations interacting with the VFS layer, is a worse option as overheads are usually higher.

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. Experimentally we found that an empty lock/unlock pair took ≈ 160 ns for user-space pthread locks and ≈ 27ns for kernel level locks on the previously described AMD Opteron. Note that 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 [17] and/or by the sender relinquishing access to the transmitted data [9]. 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.

3.1.4 Avoiding Operating System Constraints

Because these 3 constraints are difficult to manage, FShm and other recent prior work [23, 32] subvert some of these interfaces to improve performance. In particular, shared memory and concurrent lock-free data structures [13, 26] 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
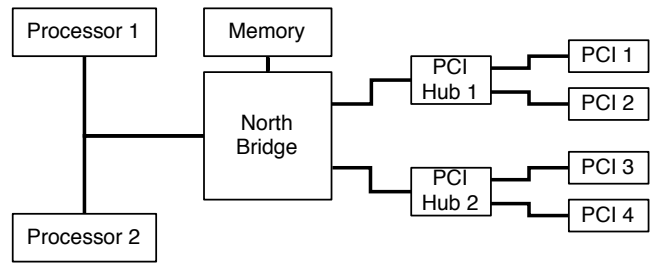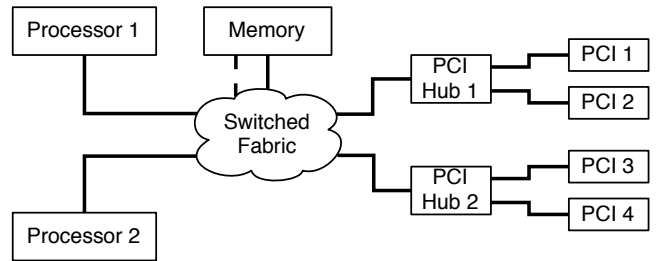

Figure 1: Host Bus Architecture

Figure 2: Switched Architecture

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 and multiple hops may be necessary since switched interconnects, such HyperTransport, are typically not implemented as full crossbars.

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 1 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 [29]. Notice that

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 1: Pentium Throughput Comparison.

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[1]. 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.

3.2.3 Handling Hardware Constraints

Fortunately, despite these constraints, we will see that with FShm modern on commodity systems can still deliver Gigabit Ethernet performance, with strong evidence that higher frame rates are possible (see Section 6). This is due, in part, to the ability of hardware prefetch units to leverage the temporal slip discussed in [13] and that ensure is data waiting in the processor's caches. Further, the situation should improve as interconnect performance improves in response to the demands of traditional applications.

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 [11, 15, 24, 31], general purpose interprocess communication and message passing [3, 5, 9, 10, 17, 30], general purpose networking on commodity hardware [14, 18, 25, 36], general purpose high-performance networking [4, 8, 10, 16, 32, 33], and concurrent lock-free data structures [13, 20, 21, 26].

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 will be examined below.

The Synthesis Kernel [24] 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

---

[1]FShm shows the same trends on these systems.

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 point-to-point CLF queue and proved its correctness under sequential consistency [12, 21], 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 [13] is a CLF queue that addresses both issues; FastForward's role in FShm is discussed in Section 5.2.

Both FShm and NetTap [4], 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.

Another related system is the nCap [8] frame processing system. 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, 4 and 5 as copies are required. Further, interface-to-interface forwarding also requires a copy.

Recently Intel has been working on ETA [33], an extension of the Virtual Interface Architecture [10] and InfiniBand [16]. The goal is to accelerate the processing of packets by dedicating a processor to act as a TCP/IP *onloading* engine [32]. This processor would preprocess packets and deliver them to applications 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 [18] 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 [22] 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 software network processor (Soft-NP) architecture. FShm leverages multicore general purpose systems with the Fast-Forward concurrent lock-free queue by Giacomoni et al. [13]
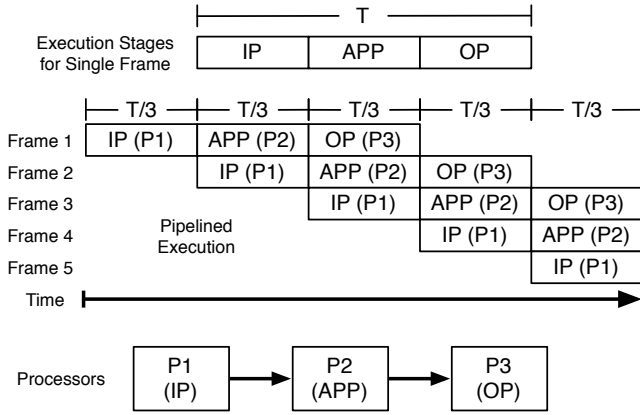
Figure 3: The Basic FShm Pipeline

Figure 4: A Basic Data-Parallel FShm Pipeline

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 specific pipeline balancing routine, a BSD mbuf compatible shared buffer wrapper, and streamline drivers. 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 discusses the streamlined drivers used by FShm. Section 5.5 concludes the design section by discussing the safety issues implicit in eliminating interprocess copy semantics.

5.1 The FShm Pipeline

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

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.
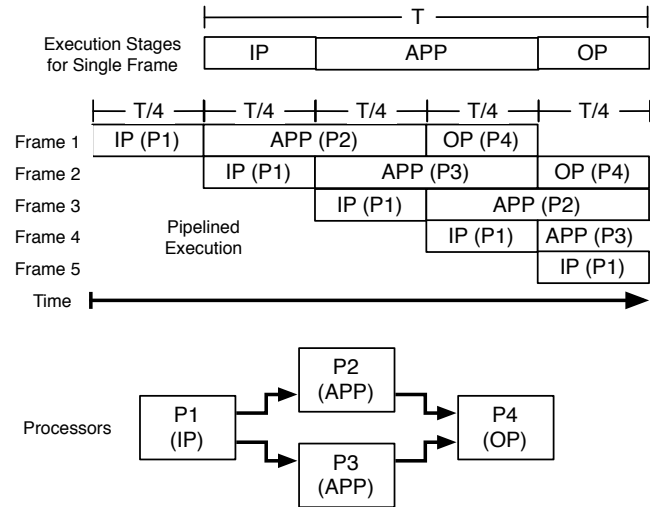
Typically, this results in triple the available per-frame processing time as the rate is externally imposed by the network. This can be achieved by permitting each stage in the pipeline to consume the full inter-frame arrival time (minus any communication overhead). Extending this pipeline to four or 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 straight forward 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 inter-frame 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 FShm 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, resulting in an unbalanced pipeline. 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, FShm uses the FastForward CLF queue [13]. FastForward provides very low overhead operations with sub-memory access latencies. Furthermore, stage-to-stage

```
1   put_nonblock(...) {
2     if (NULL == queue[head])
3     {
4       queue[head] = ptr;
5       head = NEXT(head);
6     }
7   }
```

Figure 5: CLF Queue: Put_nonblocking

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 [21] 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 cache 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 FShm, 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 [21] is realized by preventing both the putter and the getter from competing for exclusive cache access to (1) the `head` and `tail` indices, 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 FShm.

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 operating on different cachelines, thus decoupling operation as described in [13]. This has the immediate effect of increasing latency but dramatically increasing performance. Fortunately, this buffering requirement is also needed in FShm to account for interstage jitter and different frame sizes flowing through the pipeline. Therefore for the purposes of FShm, 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.

5.2.2 Automatic Pipeline Balancing

Unbalanced pipelines are present in all but the most micromanaged pipelines and must be handled efficiently. Typically FShm 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 nec-

essary 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.

Two step are necessary to ensure the stages remain balanced while processing network frames. First, each stage needs to ensure that the most of the minimum frame arrival period for each frame size is consumed, thus narrowing the stage length variations to a minimum. We accomplished this by reading the frame size and spinning on a time stamp counter (e.g., TSC in x86) to consume any additional time. Note that one should not consume the entire period as then it becomes impossible to recover from the effects of jitter causing a computation to overrun that frame's arrival period. Second, one must ensure that sufficient distance is maintained between a stage and its producer. We accomplished this by periodically (adaptively 64 to 128 frames) polling the producer's `head` index and computing the distance from the local `tail` index. If the distance is below a low watermark (i.e., dangerously close to the FastForward threshold), we spin on the TSC until the distance has increased to the desired average distance ($\approx$48 entries or 6 cachelines). Note that excessive distance unnecessarily increases latency. It is also useful to introduce a trapdoor into the spinning process to ensure forward progress is made when the line is not saturated. The loss of slip in these situations is acceptable as we have extra time between frames. These techniques were used in the evaluation.

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, the FShm drivers wrap 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 Streamlined Drivers

The drivers used in FShm are the stock FreeBSD drivers that have had a direct access API introduced and been lightly modified to remove processing best handled by the network processing application. Effort was put into maintaining general functionality of the stock drivers to allow the same driver assume either FShm behavior or normal behavior that forwards to the operating system network stack. The direct access API provides a set of interface routines that permit direct manipulation of the interface's transmit and receive descriptors from within the kernel. Further the transmit and receive interfaces were decoupled so that different modules can manipulate each set of descriptors without synchronization. It is also expected FShm wrappers will be created for intelligent interfaces providing direct user-space access and thus bypassing the standard operating system. We chose to focus supporting simple network interfaces to
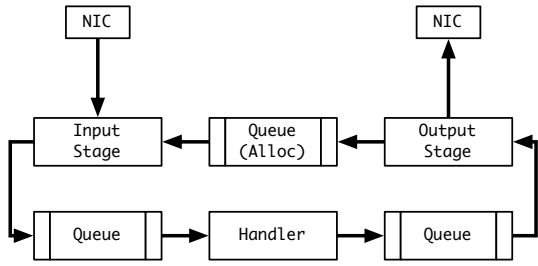
Figure 6: Evaluation Forwarding Pipeline

permit the ready use of the operating system provided stack. Additionally, the FShm drivers disable interrupts and expect the system to be polled. We plan to implement a hybrid interrupt/polled management system to conserve energy and allow for other tasks to be accomplished while the network is idle.

5.5 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 application stages that simulate 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 64bit 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. The test stages are labeled as follows: (A) for an application stage, (I) for an input stage, (O) for an output stage, and (G) for a stage encoding frames into the buffers.

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 application stages was sufficiently constant to connect three application 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 2 GHz. 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), Intel PRO/1000 (82544 EI 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 evaluation target used the two Intel cards, while the load generation host used the Broadcom card. We used two intel cards in the evaluation host to maintain experimental symmetry. 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. both cards are able to sustain approximately 1.41 mfps for 64B frames, suggesting that maybe a PCI-X or local-bus adapter related constraint for small frame sizes. For frame sizes greater than or equal to 74B, both cards were able to sustain theoretic maximum frame rates.

Second we created a test setup that had a stage (G) 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 an application stage. The application stage (A) reads the frame, spins for a specified amount of time, and then forwards it to the output stage. The output stage (O) 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. Fig-
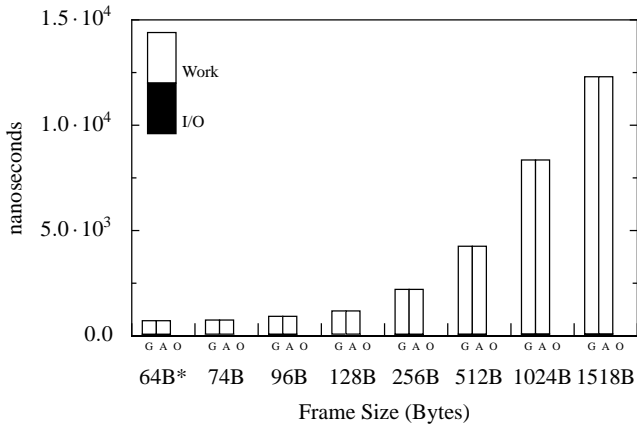
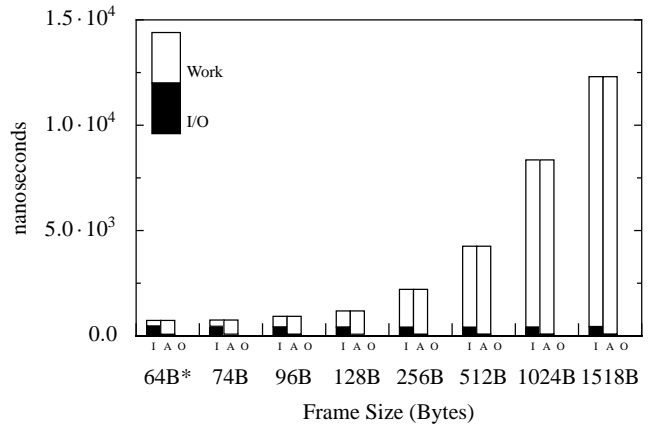**Figure 7: Generate: Available Work Time**
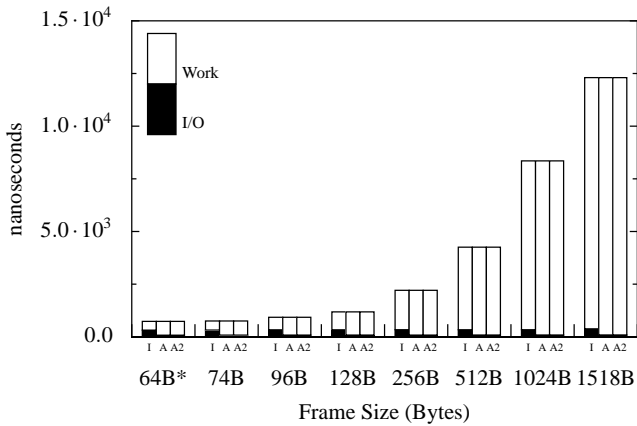

Figure 9: Forward: Available Work Time

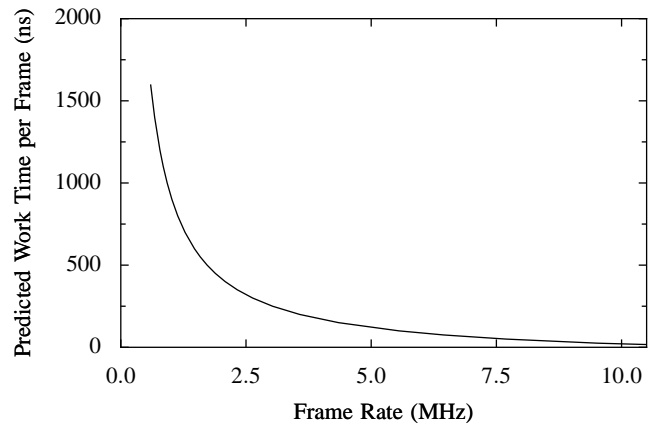

Figure 8: Capture: Available Work Time

Figure 10: Predicted: Available Work Time

ure 7 summarizes the available per-frame work times and I/O overhead for both the generation stage and the application stage, the output stage is not measured as all application processing should be complete before data is send to its output device.

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 (I) is utilized along with two application stages (A, A2). The results shown in Figure 8 show that the total available per-frame processing time in the input stage and application stages may be 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 application stages at all frame sizes, same for the input stage.

6.4 Frame Forwarding

For the frame forwarding experiments the input, application, 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 (I) and the right bar for the application stage (O). In this experiment, FShm successfully forwarded at theoretic max for 74B frames and higher. 64B frames were successfully forwarded at 1.36 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 application 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 assume 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 (OC-48 rates) as the practical cut-off for real work ($\approx$120 ns work per stage) on our aging hard-

ware platform [2]. These rates are also sufficient for realistic in-order 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 software network processor 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.

---

[2]Note that this predicted performance is actually worse than might otherwise be possible with FShm because FShm's FastForward implementation has an extra function-pointer-based abstraction layer on top of the core FastForward algorithm so that one may substitute the algorithm (e.g., to communicate with a special purpose hardware component) without recompiling.

8. REFERENCES

[1] Advanced Micro Devices. AMD completes ATI acquisition and creates processing powerhouse. `http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~113741,00.html`, October 2006.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. In *Proceedings of the twelfth ACM Symposium on Operating Systems Principles*, 1989.

[4] S. Blott, J. Brustoloni, and C. Martin. NetTap: An efficient and reliable PC-based platform for network programming, 1999.

[5] J. C. Brustoloni. Interoperation of copy avoidance in network and file I/O. In *Proceedings of the INFOCOM '99. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, 1999.

[6] Celoxia and DRC. FPGA acceleration solution released for AMD Opteron^{TM}processor-based systems. `http://www.drccomputer.com/pdfs/PR_DRC_Celoxica_042406.pdf`, April 2006.

[7] CNET News.com. Intel pledges 80 cores in five years. `http://news.com.com/2100-1006_3-6119618.html`, September 2006.

[8] L. Deri. nCap: Wire-speed packet capture and transmission. In *Proceedings of 3rd IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, 2005.

[9] P. Druschel and L. L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993.

[10] D. Dunning, G. Regnier, G. McAlpine, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 1998.

[11] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, New York, NY, USA, 1995. ACM Press.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.

[13] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *PPoPP '08: Proceedings of the The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, February 2008. ACM Press.

[14] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing extensible IP router

software. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, 2005.

[15] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*.

[16] InfiniBand Trade Association. `http://www.infinibandta.org`.

[17] Y. A. Khalidi and M. N. Thadani. An efficient zero-copy I/O framework for UNIX. Technical report, 1995.

[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 2000.

[19] S. Kumar, J. Maschmeyer, and P. Crowley. Exploiting locality to ameliorate packet queue contention and serialization. In *Proceedings of the 3rd conference on Computing Frontiers*, 2006.

[20] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. In *DISC '04: Proceedings of the Distributed Computing, 18th International Conference*, 2004.

[21] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 1983.

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

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

[24] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the twelfth ACM Symposium on Operating Systems Principles*, 1989.

[25] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings*, 1993.

[26] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. *Journal of Parallel and Distributed Computing*, 1998.

[27] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 1997.

[28] J. Mudigonda, H. M. Vin, and R. Yavatkar. Overcoming the memory wall in packet processing: hammers or ladders? In *Proceedings of the 2005 symposium on Architecture for Networking and Communications Systems*, 2005.

[29] R. Olsson. pktgen the linux packet generator. In *Proc. linuxsymposium 2005*, 2005.

[30] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-lite: A unified I/O buffering and caching system. In *Proceedings of the third Symposium on Operating Systems Design and Implementation*, 1999.

[31] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the second international conference on Architectual Support for Programming Languages and Operating Systems*, 1987.

[32] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. *IEEE Computer*, 2004.

[33] G. Regnier, D. Minturn, G. McAlpine, V. A. Saletore, and A. Foong. ETA: Experience with an Intel Xeon processor as a packet processing engine. *IEEE Micro*, 2004.

[34] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 2002.

[35] J. Verdú, J. Garciá, M. Nemirovsky, and M. Valero. Architectural impact of stateful networking applications. In *Proceedings of the 2005 symposium on Architecture for Networking and Communications Systems*, 2005.

[36] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[37] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Computer Architecture News*, 1995.

[38] B. Wun and P. Crowley. Network I/O acceleration in heterogeneous multicore processors. In *Proceedings of the 14th IEEE Symposium on High-Performance Interconnects*, 2006.