May 24, 2019

EFFICIENT QUEUE MANAGEMENT IN A PACKET BUFFER BETWEEN PROCESSES

Ian Wells

Kyle Andrew Donald Mestery

EFFICIENT QUEUE MANAGEMENT IN A PACKET BUFFER BETWEEN PROCESSES

AUTHORS:
Ian Wells
Kyle Andrew Donald Mestery

ABSTRACT

Techniques are described herein for queueing between multiple threads and processes. These techniques provide for a low-lock queue to allow multiple transmitters and receivers to successfully use a single queue efficiently. In particular, the techniques presented herein provide tactics for passing packets in a shared memory area from one process with threads to another process with a potentially different number of threads, requiring different transmitter (Tx) and receiver (Rx) queue counts on the two sides of the connection and avoiding stalls in the multiple workers as they operate on shared data structures.

DETAILED DESCRIPTION

Packets moving from one process or thread to another, for instance, from a virtual switch to a Data Plane Development Kit (DPDK) process, or from a virtual switch into a Virtual Machine (VM), are often passed in a shared memory buffer. Filling the shared memory buffer is a responsibility for the sender's threads, of which there may be many. Emptying it is a responsibility for the receiver's threads. The number of threads at the sender may not be identical to the number of threads at the receiver. This makes traditional methods, e.g., providing a number of transmitter (Tx) or receiver (Rx) queues, difficult. With conventional wired systems, the number of transmit queues on the server is irrelevant to the receiver, since the packets are serialized on the wire between the Tx queue and the Rx queue, and redivided by the receiver.

In typical virtual systems one queuing mechanism uses the same number of queues in both transmitter and receiver, with each Tx queue sharing a pointer ringbuffer with a receiver's Rx queue. This leads to an impedance mismatch between Tx and Rx processes since there is no guarantee the number of workers in the transmitter matches the number of workers in the receiver. If the number of workers differ, and particularly where there is

no common factor in those numbers, a system is required where the M input queues must accept packets and feed to N output queues to align with the worker count on either side.

Figure 1 below is a simplified block diagram of a packet queue management system, according to an example embodiment. As shown, a host computer includes two virtual machines and three Network Interface Cards (NICs). Packets that are received by the host computer at any of the NICs may be routed to any of the virtual machines through a single input memory queue. Each NIC has four writers that provide input to the memory queue, and each virtual machine has two readers that take the output form the memory queue. In general, the memory queue in a host computer has NICs providing M inputs and virtual machines providing N outputs for the queue.

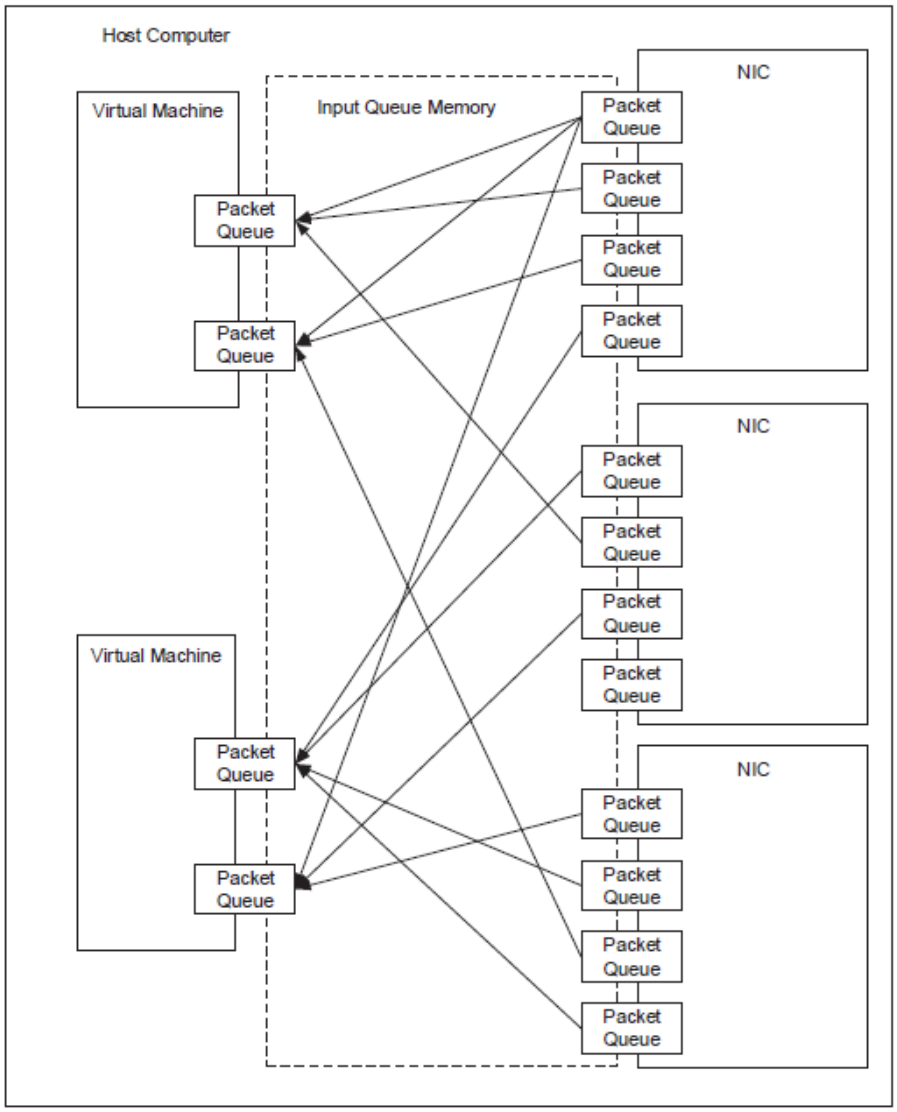

Figure 1

Figure 2 below is a diagram illustrating mapping M output queues to N input queues, according to an example embodiment. As shown, the diagram illustrates reducing the issue of coordinating M inputs with N outputs to a slightly simpler issue of mapping M inputs to a single output. This mapping may equally be viewed as M*N inputs to N outputs. Basically, the output queues need filling in such a way as to direct all packets from one flow to one queue (e.g., as may be done in Equal Cost MultiPath (ECMP) and bonded links). In one example, a hashing algorithm ensures that packets are appropriately directed during the output phase of the sending process (i.e., from the input queues of the memory). In this example, M*N notional queues, which need have no basis in reality, are directed to the N output queues. For the M queues that direct to one of the N output queues of the memory there is no specific requirement of packet ordering when those N queues feed into the single in-memory queue in the receiving application.

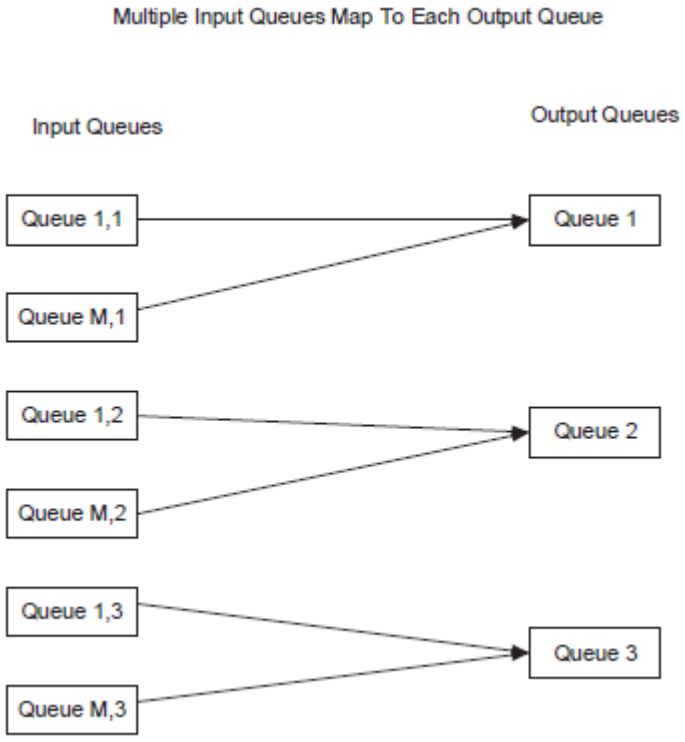

Figure 2

This resolves the receive-side issue, as each of the M input queues are mapped to each of the N output queues. However, the issue remains that M worker threads within the sender (e.g., the NICs) will all try and feed their one of the M inputs destined for queue N_x in at the same time and will contend with each other (and with the receiving application).

This may cause cache line migration between cores and will likely need spinlocks or lock-and-preempt behavior. Some examples of mitigating strategies for that contention follow.

In one example, the memory may be locked to write to output queue N_x unless otherwise stated. For instance, a spinlocks may suspend a thread such that the thread does useful work. Alternatively, a lock may cause the thread to move on to other work (e.g., putting packets in other N_x queues). In other examples, a lockless compare-and-swap may be used, although this model again has a failure mode where the swap is not performed and a determination is needed as to whether to spin on the task or move on to something else and return later.

For a particular input worker M_y, the worker threads may be selected in an order specific to M_y, which is not the same as any other of the M workers, or some reasonable approximation thereof. For instance, order of the worker threads may be selected according to a hashing algorithm. If any queue is found to be locked by another thread, the input worker M_y moves on to the next choice. Since the order is specific to that worker M_y, the next choice is made specific to My, rather than merely the next numbered queue. The worker M_y comes back to this queue after rotating through all the threads, giving the other worker (i.e., the worker that locked the memory) as much time as possible to release the lock. This should work without another worker thread jumping in, as the output phase of packet processing is a minority of the time spent processing the packet, so contention is not necessarily high. Using different queue orders for different threads assures that two threads both doing their write work will not conflict successively on every queue they would attempt to write.

In another example, the host computer avoids indirection in the memory queue. Rather than writing pointer-to-packet or consuming pointer-to-packet elements in the ring buffer, the writer threads write the packets to a block of memory reserved in the ring buffer, avoiding pointer lookups in the write process and benefitting from linear read cache behavior on the receiver. The writer threads may optionally add padding space to provide for data structures that the receiving system would construct interstitially as it performs packet processing.

Rather than a sequence of locking the buffer, writing the packet, changing the space allocation, and unlocking the buffer, which may be inefficient, the host computer may

follow a sequence of locking the buffer, consuming space for writing the packet, unlocking the buffer, writing the packet, locking the buffer, providing the space to the receiver as a full buffer, and unlocking the buffer. Both of these could potentially also be done locklessly using compare-and-swap instructions to update the buffer pointer. The receiver can use the same strategy to consume packets from the buffer.

In a further example, the host computer may move packets through the buffer memory in batches, which would be particularly efficient as regards Virtual Packet Processing (VPP). In other words, the sender may lock the buffer, consume space for ten (or another number of) packets, unlock the buffer, write all ten packets, lock the buffer, provide the buffer to the receiver, and unlock the buffer. The batch processing of the packets involves a tenth of the locking. Since the receiver may need to share the cache line with the pointers in, as well, batch processing the packets involves a tenth of the contention with the receiving core. This pattern may be mirrored at the receiver, with or without copying the buffer out of the memory space. In other words, the receiver may lock the buffer reserve incoming packet space, unlock the buffer, process the packets, lock the buffer, release the incoming memory space, and unlock the buffer.

Figures 3 and 4 below show a sequence of a writer thread passing a batch of packets to a reader thread Figure 3 is a simplified block diagram illustrating a writer thread allocating space in a shared ring buffer, according to an example embodiment. Figure 4 is a simplified block diagram illustrating a writer thread writing packets to the allocated space in the shared ring buffer, according to an example embodiment. Initially, in Figure 3, the writer thread (e.g., Writer Thread 4) has four packets to send to a reader thread (e.g., Reader Thread 2). The writer thread locks the shared ring buffer and allocates space for the batch of packets, as shown in Figure 3. Next, the writer thread unlocks the shared ring buffer and writes the four packets to the allocated space without locking the buffer, as shown in Figure 4. After the four packets are written in the shared ring buffer, the writer thread provides the full buffer as available to the reader thread.

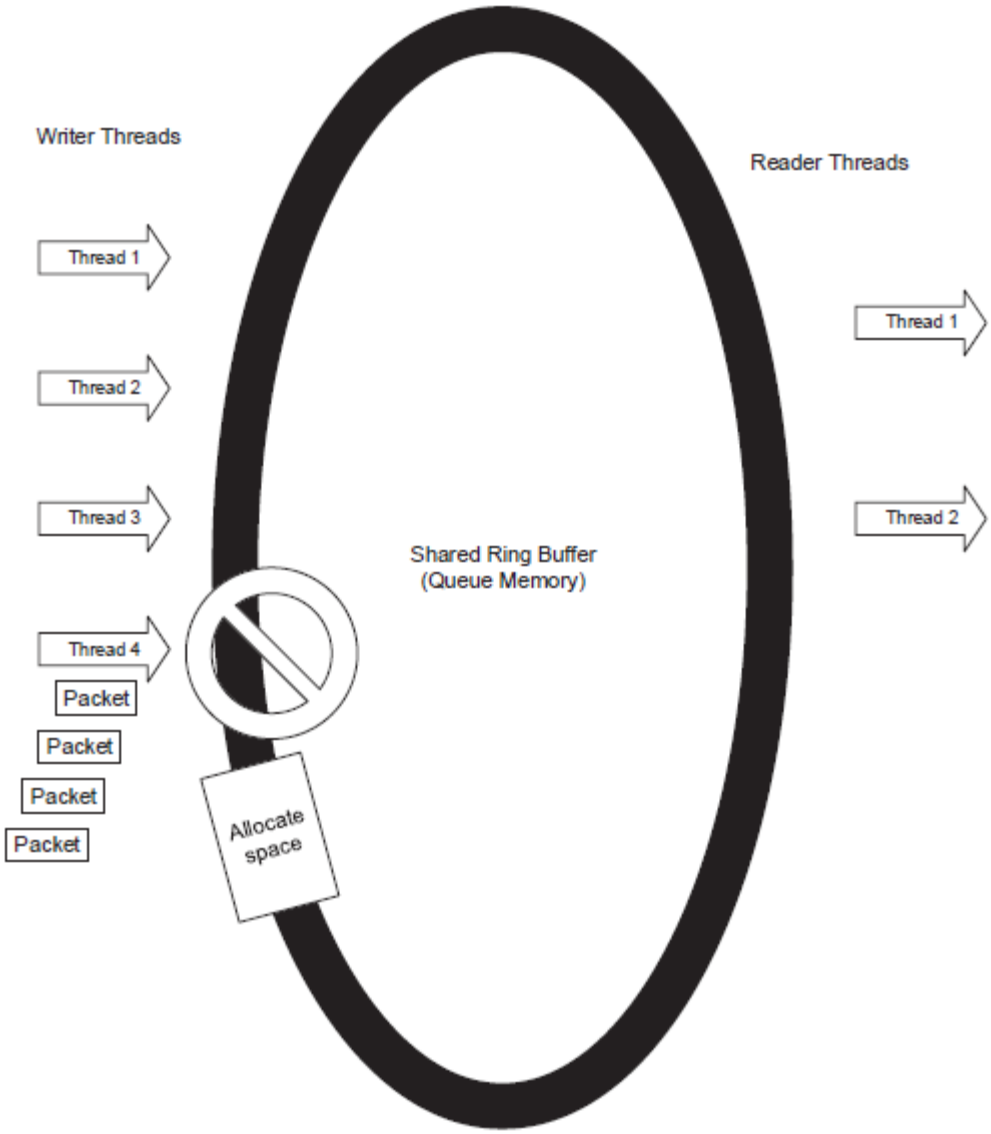
Writer Threads
Reader Threads
Thread 1
Thread 2
Thread 3
Thread 4
Thread 1
Thread 2
Shared Ring Buffer
(Queue Memory)
Packet
Packet
Packet
Packet
Allocate space

Figure 3

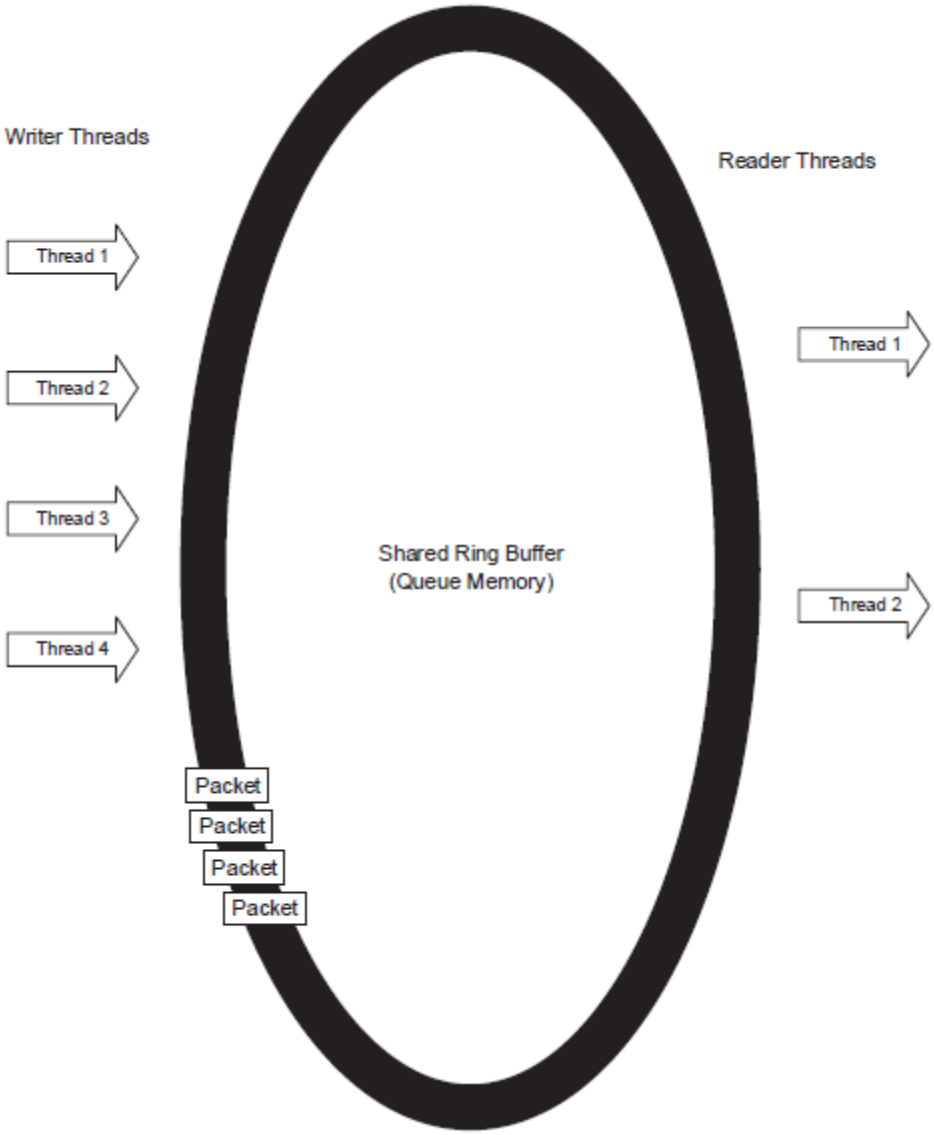

Figure 4

In one example, two threads writing a packet or packets to a queue may hand over their space to the receiver out of order, which would not work in a ring buffer model. To resolve this issue, a semaphore may be updated to show a receiver what packets are available only when all threads release their data, and not until all threads are ready. This could potentially starve the receiver of packets since no packets would be released until all threads had finished, but on the other hand the work in the locked section is purely a memory copy and is being run concurrently by multiple cores, which (subject to the von Neumann limit) is likely to be parallelizable.

Pointers and indirection can still be used and may offer a benefit. If the receiver can be allocated memory from a buffer that is not necessarily in-order, then the memory blocks can be released to the receiver in any order, since the ordering of data provided from different M threads is not required, per flow rules described herein. Using points/indirection may or may not be efficient for the receiver based on the data plane inside the receiver but it does avoid starvation.

In another example, the sender and receiver may use one-side-write memory. In conjunction with the batch processing mechanism, a one-side-write mechanism may avoid cache line contention between the senders and the receiver. This may be more efficient and avoids issues with the sender and receiver both being able to write the same area of memory, leading to debug mysteries when there is any sort of problem based on not knowing which side wrote a value to the memory, as well as more limited (and therefore detectable or predictable) failure types from the reading side.

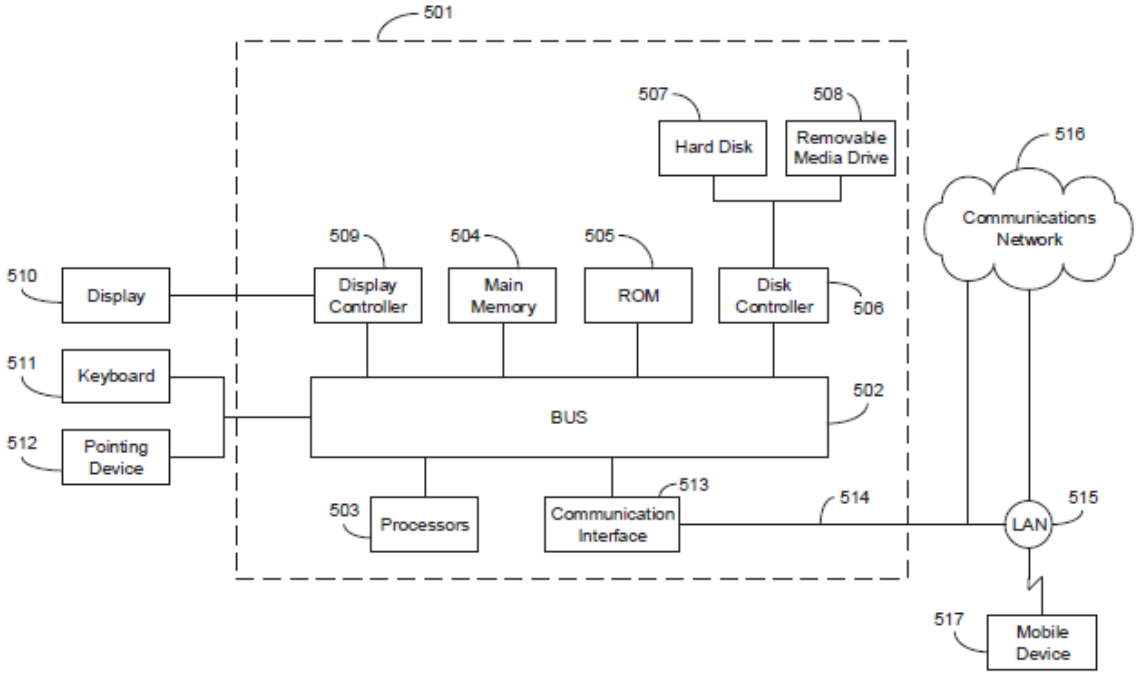

Figure 5

Referring now to Figure 5 above, shown is a simplified block diagram of a device that may be configured to perform methods presented herein, according to an example embodiment. Figure 5 illustrates an example of a block diagram of a computer system 501 that may be representative of the host computer in which the embodiments presented may

be implemented is shown. The computer system 501 may be programmed to implement a computer based device, such as a laptop computer, desktop computer, tablet computer, smart phone, internet phone, network element, or other computing device. The computer system 501 includes a bus 502 or other communication mechanism for communicating information, and a processor 503 coupled with the bus 502 for processing the information. While the figure shows a single block 503 for a processor, it should be understood that the processors 503 may represent a plurality of processing cores, each of which can perform separate processing. The computer system 501 also includes a main memory 504, such as a random access memory (RAM) or other dynamic storage device (e.g., dynamic RAM (DRAM), static RAM (SRAM), and synchronous DRAM (SD RAM)), coupled to the bus 502 for storing information and instructions to be executed by processor 503. In addition, the main memory 504 may be used for storing temporary variables or other intermediate information during the execution of instructions by the processor 503.

The computer system 501 further includes a read only memory (ROM) 505 or other static storage device (e.g., programmable ROM (PROM), erasable PROM (EPROM), and electrically erasable PROM (EEPROM)) coupled to the bus 502 for storing static information and instructions for the processor 503.

The computer system 501 also includes a disk controller 506 coupled to the bus 502 to control one or more storage devices for storing information and instructions, such as a magnetic hard disk 507, and a removable media drive 508 (e.g., floppy disk drive, read-only compact disc drive, read/write compact disc drive, compact disc jukebox, tape drive, and removable magneto-optical drive, solid state drive, etc.). The storage devices may be added to the computer system 501 using an appropriate device interface (e.g., small computer system interface (SCSI), integrated device electronics (IDE), enhanced-IDE (E-IDE), direct memory access (DMA), ultra-DMA, or universal serial bus (USB)).

The computer system 501 may also include special purpose logic devices (e.g., application specific integrated circuits (ASICs)) or configurable logic devices (e.g., simple programmable logic devices (SPLDs), complex programmable logic devices (CPLDs), and field programmable gate arrays (FPGAs)), that, in addition to microprocessors and digital signal processors may individually, or collectively, include types of processing circuitry. The processing circuitry may be located in one device or distributed across multiple devices.

The computer system 501 may also include a display controller 509 coupled to the bus 502 to control a display 510, such as a cathode ray tube (CRT), liquid crystal display (LCD) or light emitting diode (LED) display, for displaying information to a computer user. The computer system 501 includes input devices, such as a keyboard 511 and a pointing device 512, for interacting with a computer user and providing information to the processor 503. The pointing device 512, for example, may be a mouse, a trackball, track pad, touch screen, or a pointing stick for communicating direction information and command selections to the processor 503 and for controlling cursor movement on the display 510. In addition, a printer may provide printed listings of data stored and/or generated by the computer system 501.

The computer system 501 performs a portion or all of the processing steps of the operations presented herein in response to the processor 503 executing one or more sequences of one or more instructions contained in a memory, such as the main memory 504. Such instructions may be read into the main memory 504 from another computer readable storage medium, such as a hard disk 507 or a removable media drive 508. One or more processors in a multi-processing arrangement may also be employed to execute the sequences of instructions contained in main memory 504. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions. Thus, embodiments are not limited to any specific combination of hardware circuitry and software.

As stated above, the computer system 501 includes at least one computer readable storage medium or memory for holding instructions programmed according to the embodiments presented, for containing data structures, tables, records, or other data described herein. Examples of computer readable storage media are compact discs, hard disks, floppy disks, tape, magneto-optical disks, PROMs (EPROM, EEPROM, flash EPROM), DRAM, SRAM, SD RAM, or any other magnetic medium, compact discs (e.g., CD-ROM, DVD), or any other optical medium, punch cards, paper tape, or other physical medium with patterns of holes, or any other medium from which a computer can read.

Stored on any one or on a combination of non-transitory computer readable storage media, embodiments presented herein include software for controlling the computer system 501, for driving a device or devices for implementing the operations presented

herein, and for enabling the computer system 501 to interact with a human user (e.g., a network administrator). Such software may include, but is not limited to, device drivers, operating systems, development tools, and applications software. Such computer readable storage media further includes a computer program product for performing all or a portion (if processing is distributed) of the processing presented herein.

The computer code devices may be any interpretable or executable code mechanism, including but not limited to scripts, interpretable programs, dynamic link libraries (DLLs), Java classes, and complete executable programs. Moreover, parts of the processing may be distributed for better performance, reliability, and/or cost.

The computer system 501 also includes a communication interface 513 coupled to the bus 502. The communication interface 513 provides a two-way data communication coupling to a network link 514 that is connected to, for example, a local area network (LAN) 515, or to another communications network 516 such as the Internet. For example, the communication interface 513 may be a wired or wireless network interface card to attach to any packet switched (wired or wireless) LAN. As another example, the communication interface 513 may be an asymmetrical digital subscriber line (ADSL) card, an integrated services digital network (ISDN) card or a modem to provide a data communication connection to a corresponding type of communications line. Wireless links may also be implemented. In any such implementation, the communication interface 513 sends and receives electrical, electromagnetic or optical signals that carry digital data streams representing various types of information.

The network link 514 typically provides data communication through one or more networks to other data devices. For example, the network link 514 may provide a connection to another computer through a local area network 515 (e.g., a LAN) or through equipment operated by a service provider, which provides communication services through a communications network 516. The local network 514 and the communications network 516 use, for example, electrical, electromagnetic, or optical signals that carry digital data streams, and the associated physical layer (e.g., CAT 5 cable, coaxial cable, optical fiber, etc.). The signals through the various networks and the signals on the network link 514 and through the communication interface 513, which carry the digital data to and from the computer system 501 may be implemented in baseband signals, or carrier wave based

signals. The baseband signals convey the digital data as unmodulated electrical pulses that are descriptive of a stream of digital data bits, where the term "bits" is to be construed broadly to mean symbol, where each symbol conveys at least one or more information bits. The digital data may also be used to modulate a carrier wave, such as with amplitude, phase and/or frequency shift keyed signals that are propagated over a conductive media, or transmitted as electromagnetic waves through a propagation medium. Thus, the digital data may be sent as unmodulated baseband data through a "wired" communication channel and/or sent within a predetermined frequency band, different than baseband, by modulating a carrier wave. The computer system 501 can transmit and receive data, including program code, through the network(s) 515 and 516, the network link 514 and the communication interface 513. Moreover, the network link 514 may provide a connection through a LAN 515 to a mobile device 517 such as a personal digital assistant (PDA), tablet computer, laptop computer, or cellular telephone.

The techniques described herein are not necessarily specific to VMs, though today most Virtual Network Function (VNF) workloads are virtual machines. They may also apply to processes or systems that use multiple threads to perform tasks, and are applicable to kernels if they are in a VM and used an interface of this nature, as well as Data Plane Development Kit (DPDK)-style passthrough processes in VMs, containers, or bare metal.

In all of these cases, it is most efficient within a process to have one input queue per thread such that it is not fighting for packets and locks with the other processes as it performs a receive. This may also prompt the input queue to become tied to a single physical Central Processing Unit (CPU), and also create one output queue per thread. In a physical system with a single process running, the number of input and output queues of the NIC matching the thread count may be used. In a physical system, the queues in the NIC feed are fed from a physical wire in hardware, and the N-to-1 / 1-to-N process is handled by hardware and thereby avoids software-based locking and contention problems. However, two processes that are communicating over a shared memory interface are not guaranteed to have the same number of worker threads performing packet processing, which can create queue count mismatch (M != N). This is not a problem with draining, but rather a problem with the optimal queue count for either side of the interface. This is

reduced to achieving some level of efficiency as the packet is moved from M threads to N queues on output.

In summary, techniques are described herein for queueing between multiple threads and processes. These techniques provide for a low-lock queue to allow multiple transmitters and receivers to successfully use a single queue efficiently. In particular, the techniques presented herein provide tactics for passing packets in a shared memory area from one process with threads to another process with a potentially different number of threads, requiring different transmitter (Tx) and receiver (Rx) queue counts on the two sides of the connection and avoiding stalls in the multiple workers as they operate on shared data structures.