Zero-Copy TCP in Solaris

H. K. Jerry Chu
SunSoft, Inc.

Zero-Copy TCP in Solaris

Hsiao-keng Jerry Chu
SunSoft Inc.

Abstract

This paper describes a new feature in Solaris that uses virtual memory remapping combined with checksumming support from the networking hardware, to eliminate data-touching overhead from the TCP/IP protocol stack. By implementing page remapping operations at the right level of the operating system, and caching MMU mappings to take advantage of locality of reference, significant performance gain is attained on certain hardware platforms. Nevertheless, the performance improvement over CPU copying varies, depending on the host memory cache architecture, MMU design, and application behavior. We begin by comparing different zero-copy schemes, and explain our preference for page remapping and copy-on-write (COW) techniques. We then describe our implementation, and present its performance characteristics under a number of different parameters. We conclude with ideas for future improvements.

1. Introduction

High data throughput is one of the major requirements for applications such as multimedia and video-on-demand operating over high-speed networks like ATM. Networking software or hardware is often a major bottleneck in this respect, and considerable research focuses on delivering the vast bandwidth effortlessly.

Analysis of components of networking software reveals that data copy and checksum overhead dominates processing time for high throughput applications [3, 10, 17]. Older generation networking software and hardware often required multiple data copy and separate data checksum operations on each byte of a data packet. The past few years have seen a number of successful implementations (such as in Solaris release 2.4) introducing "*single-copy*" (CPU copy). They are often combined with the TCP checksum calculation in one single loop, resulting in substantial throughput improvement[17].

Recent efforts focus on designing an optimal architecture capable of moving data between application domains and network interfaces without CPU intervention, in effect achieving "*zero-copy*[*]". Issues that arise are architecture efficiency, cost, application and driver programming interfaces, software complexity, and compatibility. Proposals attaining the best efficiency often deviate from the existing UNIX networking I/O interface, or may rely on special hardware. As developers of a widely used operating system, we were resolved not to alter any interface, either at application or device driver level[†]. There is a considerable installed base of software investment to protect. Further, for broader market appeal we felt it unwise to rely on specialized hardware, as some of the cited examples in literature did.

We turned to virtual memory remapping and copy-on-write technique, both being widely adopted in operating system design to avoid copying [1, 2, 8, 16]. Although these operations are not without expense, they often can be applied transparently, and so fit our goal of minimizing software module change. Networking adaptors used are Sun's SBus-based ATM interface cards, capable of both 155 and 622 Mb/s, OC-3 and OC-12 respectively.

In the next section, we illustrate the importance of reducing data touching overhead as packet size increases in new generations of network media. In Section 3 we introduce zero-copy, and describe several different approaches. Our implementation based on page remapping and copy-on-write is described in detail in Section 4. Section 5 examines aspects of host architectures and their impact on the performance of zero-copy and single-copy, to set the stage for the measurement results presented in Section 6. Section 7 presents our conclusions.

---

*. This is from the CPU point of view. A bus centric view would count DMA transfer as one copy.

†. Some additions to the driver interface were required to support hardware checksum though.

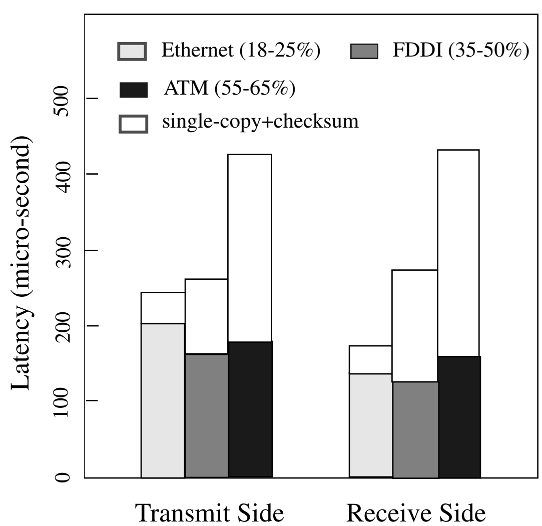
2. Networking Software Overhead

The overhead in networking software can be broken up into per-packet and per-byte costs. Generally, the per-packet cost is roughly constant for a given network protocol, regardless of the packet size, whereas the per-byte cost is determined by data copying and checksumming overhead. To reduce per-byte cost, Solaris 2.4 introduced combined single copy and data checksum design.

High throughput applications often send large packets to amortize per-packet costs over many data. However, the communication link imposes an upper bound on the packet size that can be accepted. This limit (called maximum transmission unit or MTU) is relatively small for traditional network media (1500 bytes for Ethernet). Therefore, the communication overhead on those networks is still dominated by the per-packet cost.

Newer generation of communication media employ larger MTUs – it is 4352 bytes for FDDI, 9180 bytes for IP over ATM, and up to 64KB for other protocols over ATM. The per-byte cost on these networks adds up to a significant portion of the total networking overhead, as demonstrated by the following chart.

Figure 1: Networking overhead in Solaris 2.5.



The figure shows per-byte (single-copy+checksum) costs as a percentage of total networking software costs for a MTU size packet, measured on SPARCstation/20 running memory-to-memory TCP tests over three different types of network.

For TCP/IP packets going through ATM networks, the copy+checksum overhead accounts for over 60% of the total networking software overhead on the end hosts. Our goal is to further reduce the per-byte cost from the current single-copy+checksum design.

3. Zero Copy + Hardware Checksum

This section offers some insight on requirements involved in avoiding copy and checksum in networking software. We also evaluate various approaches seen in the literature.

3.1. Disk I/O versus Network I/O

Zero-copy I/O already exists in UNIX – neither memory mapped file nor raw disk I/O require any memory copy. The former offers an efficient way of accessing static, mappable objects, but is not applicable to a pipe-like network data flow. The difference between raw disk I/O and network I/O is more subtle, yet substantial. The former is synchronous in nature, while the latter is highly asynchronous.

We first discuss the write side. A raw disk write is initiated synchronously in the same program context that issues the write request. This implies the user buffer can be accessed directly by the disk driver.

The write request will block until the disk controller finishes the data transfer, which usually takes only milliseconds. Nevertheless, network I/O may take longer to really "complete" because packets may get lost somewhere in the network.

To guarantee data delivery to the destination host, a reliable data delivery protocol such as TCP must retain user data for possible retransmission. UNIX network software handles this by copying user data into a kernel buffer, and queueing it to the right outbound queue. Then it returns to the user program immediately without waiting for data transmission and acknowledgment to finish. This behavior is important for achieving high data throughput, since an application can post multiple data packets simultaneously to fill a long network pipe. Another implication of these copy semantics is that an application may safely reuse the same buffer to prepare data for the next transfer.

To achieve zero-copy, as in raw disk I/O, virtual memory (VM) operations can be used to lock down user memory and map it into kernel address space. This makes user data accessible to the protocol software and the device driver, even after the user program returns. But if the application reuses the same buffer too soon, it may destroy previous data still waiting to be sent, unless measures are taken to protect the user buffer before the transport layer is finished with it.

For a raw disk read, the read buffer is posted first, followed by disk I/O. Therefore, before the disk controller starts the data transfer, it already knows where to put the data.

In the networking case, packets arrive asynchronously at the network interface. There may or may not be read buffers posted from the receiving clients. Limited interface memory requires newly arrived data to be transferred into regular memory quickly to free up resources. Even if read buffers are posted, special preview of packet headers by the hardware is required to determine which read buffer to place the packet, as there are often multiple clients, each has a different read buffer.

3.2. Different Zero Copy Schemes

Several zero-copy schemes have been proposed in the literature. Smith et al. [13] outlines a variety of approaches to host interface design and supporting software. Steenkiste [14] presents a taxonomy of host interfaces and their numbers and types of data movement across a memory bus. In the following, we classify all the approaches into four categories, and briefly describe the pros and cons for each.

3.2.1 User accessible interface memory

The best scenario with minimal data transfer overhead is one in which the network interface memory is accessible and pre-mapped into user and (possibly) kernel address space. Data never needs to traverse the memory bus until is accessed by the application.

Unfortunately, this requires complicated hardware support and substantial software changes. For one thing, cache consistency has to be maintained either through software flushing or special hardware arrangement. On the receive side, to avoid remapping memory, it requires intelligence in the network hardware to direct incoming data to the right interface memory pool, since the interface memory is likely to be shared among multiple clients.

Obviously, this scheme also requires applications to use special buffer management calls to allocate and use the interface memory. Software compatibility and portability do not exist.

Limited interface memory could pose a serious resource problem. Memory hogs or bug-ridden applications with memory leaks can easily deplete the interface memory available for use [15].

A variant of this approach that uses a dedicated coprocessor for protocol processing is described by Cooper et al [4].

3.2.2 Kernel–network shared memory

To alleviate the resource problem described above, this scheme lets the operating system kernel manage the interface memory, and uses direct memory access (DMA), or program I/O (PIO, i.e. CPU copy), to move data between interface memory and application buffers. Data only travels across the memory bus once (DMA) or twice (PIO), so overhead is minimized. A further advantage lies in existing applications not being required to undergo modifications, since the socket's copy semantic is fully maintained by this approach. *Afterburner*, a classic example in this category, is described by Dalton et al. in [5].

The software to support this scheme can be complicated. Kernel networking buffer management code must be enhanced to support this special pool of memory from the network interface, which it co-manages with the device driver. With TCP checksum being performed during DMA or PIO, it poses a problem on the receive side. Packets cannot be verified and acknowledged by TCP until receiving clients run and post read requests. If this doesn't happen in time, the transmit side will time-out and generate unnecessary retransmissions.

If DMA is used, it requires pinning and unpinning of user pages, and possibly mapping from the kernel context, which add to the costs [11].

Applications can no longer hog the interface memory directly, but TCP retransmit buffers still reside in the interface memory. Even though flow control exerted by TCP and the socket layer together impose an upper limit on the amount of memory each connection may consume, a few hung TCP connections can still starve interface memory.

3.2.3 User–kernel shared memory

This scheme defines a new set of application programming interfaces (APIs) with shared semantics between the user and kernel address spaces, and uses DMA to move data between the shared memory and the network interface. One proposal in this category is called *fast buffers (fbufs)* described by Druschel and Peterson in [6]. It uses a per-process buffer pool that is pre-mapped in both the user and kernel address spaces, thus eliminating the user–kernel data copy. Ideally, each data byte crosses the memory bus exactly once, so overhead is low and no special, processor-addressable interface memory is needed.

One major disadvantage of this approach is application compatibility. All the programs have to be converted to use this alternate set of APIs and programming model.

Managing the shared buffer pool requires close cooperation between the application, networking software, and the device driver, all allocating memory from the same pool. On the receive side, virtual address and physical memory fragmentation may develop, either because the application needs to hold on to some data while relinquishing others, or when TCP handles out-of-order or duplicate packets, or when the driver passes up partially filled buffers due to smaller-than-MTU packets arriving from the network. Fragmentation not only makes the application programming model more complicated, but also may pose problems to DMA engines, as many of them have special size or alignment requirement. For the latter, it may be necessary to resort to copying.

On the receive side, network hardware must be capable of targeting DMA transfer of an incoming packet to the correct memory pool allocated by the receiving client.

Due to the nature of shared memory, an error-prone application may inadvertently modify memory contents previously sent, causing data corruption problems that are hard to debug. The shared memory model also makes it difficult to connect with other types of memory objects, as close cooperation is required of other memory managers.

3.2.4 User−kernel page remapping+COW

This scheme uses DMA to transfer data between interface memory and kernel buffers, and remaps buffers by editing the MMU table to give the appearance of data transfer. By combining it with COW (copy-on-write) technique on the transmit side, it preserves the copy semantics of traditional socket interface. It also connects well with the rest of the VM system and other types of memory objects.

This approach is not without shortcomings, however. The VM operations can be expensive. All the buffers involved must align on page boundaries, and occupy an integral number of MMU pages. On the transmit side, this is less of a problem, as any fragmented part of a user buffer can be transmitted separately, using CPU copy. On the receive side, this can be problematical. First, packets must be large enough to cover a whole page of data. This limits the scheme to networks with an MTU size larger than the system page size, such as FDDI and ATM. Secondly, in order to apply page remapping to a stream of packets without disruption, user payload of all the packets must contain an integral number of pages. This can be accomplished by TCP negotiating a maximum segment size (MSS) of such a size. Thirdly, network drivers must arrange receive buffers in such a way that, after DMA, user payload shows up on a page boundary in the buffer. Without trailer encapsulation, driver software has to accurately predict the size of protocol headers to skip. For protocol headers of a fixed length, such as TCP/IP (assuming no options), this is easy. But for NFS and similar protocols where the header length varies depending on requests, one is forced to choose a size that works for a majority of cases. One solution requires adding special hardware logic to the DMA engine to parse packet headers, so that it knows where to split the data from the headers on the fly.

On the transmit side, even though data is write-protected from the application during transfer, applications still should avoid reusing busy buffers because copy-on-write faults can be very expensive. Without an end-to-end level acknowledgment, applications can not tell whether a buffer has been released by the transport or not. Fortunately, the networking software imposes an upper bound on how much data can be outstanding before flow control is exerted. For example, in the current TCP implementation in Solaris, the maximum socket send buffer size plus TCP window size totals to 128K. Therefore, if an application uses a circular list of buffers with total size greater than 128K, it will block before it has a chance to overwrite any busy buffer.

Once the alignment requirement is met, most of the software work is confined to the VM system. Very little change is needed in the networking code, except at the socket layer to replace *copyin()* with *mapin+cow*, and *copyout()* with *remap*. This scheme then becomes most attractive for us.

3.3. Hardware Checksum

All the approaches described above, with the exception of PIO, call upon special hardware to calculate data checksums, usually during DMA transfers. This is particularly important for zero-copy, as the cost of CPU checksum can become more conspicuous on a cold cache resulted from zero-copy. For PIO (single-copy), hardware checksum is less critical, since folding checksum calculation into the copy loop often makes the cost of software checksum negligible [17].

For protocols that store the checksum in the header, (e.g. TCP), it poses a problem on the transmit side for interface memory that is organized as a FIFO. In that case, a separate checksum pass over the data is required, unless a trailer checksum is used.

Note that pushing the data checksum calculation to the interface hardware weakens its protection against data corruption. Specifically, data corruption over I/O bus will not be caught. However, the extremely low error rate in a modern I/O bus, as well as other data transfers over the I/O bus for common devices like disks, are routinely assumed to be correct and not checked in software. We therefore conclude that such a reduction in protection is not unreasonable.

4. Design and Implementation

In this section, we outline our design goals, constraints, and briefly discuss our implementation.

4.1. Design Goals

- Fit the design into Solaris VM architecture [9, 12]. Build the new remapping + copy-on-write functionality on top of the existing VM base.
- Implement the new functionality outside of VM segment layer and vnode layer. Thus it can work transparently on any type of memory objects applications choose to use. In particular, it should support memory mapped files as well as anonymous memory.
- Minimize changes to the existing VM code to minimize impact on the current system's stability and performance.

4.2. Constraints

Constraints involved in pursuing our goals were:

- Avoid creating new interfaces, or altering the existing ones. This includes both the application level and the device driver level.
- Existing applications and drivers should benefit with minimal changes (e.g. buffer alignment).

4.3. Implementation

The key to attaining efficiency in the implementation is to take advantage of information already loaded in the MMU tables. For a user buffer in an address space, the physical page behind it and its protection mode can be quickly retrieved from the MMU if the buffer is currently mapped. This is much faster than going through the VM system using the regular top-down path. Also two new HAT[‡] layer operations, *hat_softlock()* for the transmit side, and *hat_pageflip()* for the receive side, were invented to perform all the MMU operations in one call. This is much faster than calling multiple existing HAT operations to achieve the same task.

Another key to code efficiency is to focus on the fast path where the majority of cases execute, and fall back to copy for the rest. For example, the code acquires locks and manipulates VM data structures in a way that is most efficient for what it tries to accomplish. But in some cases it violates the locking order defined by the VM system. In order to avoid deadlocks, when the code discovers that a lock it tries to acquire is held already, instead of waiting for the lock to be released, it simply backs off and takes the copy path.

On the transmit side, when entering the write system call, the user buffer has to be mapped into the kernel address space. Since an application often allocates and reuses the same buffer, a per-process cache is created to keep around kernel mappings created. If the whole user buffer fits into the kernel mapping cache, after the first pass, no more map-in operation is needed. This also helps the performance considerably.

4.4. Operation Steps

In the following, we step through the major operations on both the transmit and the receive side. With the exception of the transport calling back when a packet is acknowledged, all operations are performed at the Stream head.[**]

4.4.1 Transmit side

When entering the write system call -

1. Call *hat_softlock()* to locate the page behind a user buffer and lock it down.
2. Check into the kernel mapping cache to see if there is a cached kernel mapping to the same user page from previous requests. If so, simply reuse its kernel address. Otherwise, map in the user page.
3. Change the user protection to read-only. Mark the user page as copy-on-write page.

When the transport is finished with the user page -

1. Unlock the page.

---

‡. Hardware Address Translation. See [9].

**. Solaris networking subsystem uses AT&T STREAMS framework.

2. Restore the user protection back to read-write. Tear down the copy-on-write state.
3. Do NOT unmap and deallocate the MMU resources of the kernel mapping. Cache it for possible future use.

4.4.2 Receive side

Inside the read system call -

1. Given a user buffer and a kernel buffer, call *hat_pageflip()* to look up and hold exclusive locks on both pages behind the addresses. Then flip two sets of MMU translations all in one function.
2. Fix all the VM data structures to reflect new page identities and vnode associations (*page renaming*).
3. Unlock the new data page. Give the old user page back to the driver.

5. Performance Evaluation

How well zero-copy performs against single-copy is mainly determined by how efficiently page remapping and COW operations can be made relative to memory copy. Both are heavily dependent upon the underlying machine architecture. In this section, we first examine the aspects of the host architecture that affect their operation efficiency, and describe architecture details of various Sun workstations we use. We then describe our software and hardware setup for performance measurements.

5.1. Page Remapping+COW Overhead

Most software overhead is incurred in the VM system, such as looking up memory pages and updating data structures. More significant is hardware-related overhead, such as reprogramming the MMU and flushes to keeping various caches consistent.

To remap each page, the software must

- flush the obsolete entry from the local translation look-aside buffer (TLB).
- flush stale data from the local data cache if the cache is virtually addressed. Note that for a virtual-indexed physical-tagged (VIPT) cache, such as the one used in HyperSPARC, flushes can be deferred if the new mapping has the same virtual color as the old one.
- On multiprocessor (MP) machines, flushes need to be repeated on remote CPUs. Unless special hardware is employed, software cross-call is often used to interrupt remote CPUs to perform the flush.

To set up a copy-on-write protection, one needs to

- change the MMU protection to disallow write, and flush the old TLB entry.
- flush dirty data from the local cache if the cache is virtually addressed[††].
- on MP machines, repeat the above flushes on remote CPUs.

To tear down copy-on-write protection, one needs to

- restore the original user protection mode in the MMU, and flush TLB entry again if necessary.

All the cache and TLB flushes above may inflict some penalty upon the applications that try to access the data later. The table below lists some hardware characteristics of the machines we use for performance measurement.

Table 1. Host Architectures

	Processor	E-cache (all copy-back)	Remote tlb flush
SPARCstation/20	60MHz SuperSPARC	1MB physical address	software xcall
SPARCstation/20-HS11	100MHz HyperSPARC	256KB VIPT	software xcall
SPARC-server-1000E	60MHz SuperSPARC	1MB physical address	XBus broadcast

5.2. CPU Copy Overhead

Modern computer architectures use sophisticated hardware caches to speed up CPU access to memory, based on the principle of locality of reference. Moreover, some advanced workstations employ sophisticated hardware to move data between different memory locations without CPU intervention. The latter has a few different variations with respect to keeping the cache consistent with the main memory, which may affect application performance. For example, the latest UltraSPARC processor offers special block load and store instructions that do not allocate in the E-cache on a miss. Therefore, block copy will NOT produce hot cache data on destination unless destination lines reside in the cache before copy.

For memory copy using regular load/store instructions, the speed is determined by the cache state of the data, both source and destination, and the effectiveness of the memory cache hierarchy. With each level

††. Strictly speaking, the flush is not needed for a VIPT cache. It is currently done to simplify cache aliases handling.

of cache misses, access latency to the next level increases dramatically. A potential benefit of software memory copy is that it produces a warmer cache if data is consumed soon after the copy. On the other hand, excessive copy operations may cause cache thrashing, and impair overall system performance.

The following table shows CPU copy latencies on uniprocessor (UP) SPARCstation/20 and 8-way SPARCserver/1000E. Both machines use 60MHz SuperSPARC processors with small on-chip cache, and large (1MB) second level external cache. Note that the E-cache controller (MXCC) offers hardware block copy function. But it is only used in kernel-to-kernel memory copy, and is not applicable here.

Table 2. CPU copy latency (micro-second)

Copy 8192 bytes	SS20	SS1000E
Ehot+Dhot	37	37
Ehot+Dcold	62	62
Ecold+Dcold	257	345
Edirty+Dcold	330	388

The cache states on the first column apply to both the source and the destination cache lines. For example, the last row shows the latency numbers when both the source and destination have cache misses, and the target cache lines are dirty. In this case, the dirty data needs to be flushed back to main memory before the new data can be fetched. This explains why it takes longer than the previous row.

The cache hit numbers are the same on two machines, as both use the same type of processor and E-cache. The last two rows show that SPARCstation/20's memory system is faster than SPARCserver/1000E's.

5.3. DMA Architecture

Sun's SBus adopts a virtual address based DMA (DVMA) design, where a DMA address is first translated through an I/O memory management unit (IOMMU) before used. Driver software on the host is responsible for setting up IOMMU mappings before starting a DMA transfer.

In most of the recent Sun workstations, main memory cache coherency with DMA I/O is maintained automatically by the SBus to memory bus interface logic. Therefore, no CPU flush is needed. On machines with virtual address caches (e.g. SPARCstation/20-HS11), additional cache alias rule is required. That is, the DVMA address and the corresponding host memory address must map to the same cache lines. Otherwise, explicit cache flushes by the processor are required.

All the memory buses employ a write invalidate protocol for maintaining cache coherency. This has performance ramifications on the receive side—after a DMA transfer, the data will not be in the cache. When the processor accesses the data later, the data must be loaded from main memory into the cache first.

5.4. ATM Host Adaptors

We use Sun's SBus ATM interface card "*SunATM adaptor 2.0*" which supports both 155 and 622 Mb/s ATM interfaces, with a packet level interface to the host that hides the details of ATM cells. DVMA is used to move data between host memory and interface FIFO buffers, with a per-receive channel programmable header size for header/data splitting.

On the transmit side, TCP checksum calculation is done on the fly during DMA data transfer from host memory to interface memory. An entire packet is held in the interface buffer before segmentation, so that the checksum result can be stuffed into the TCP header. On the receive side, checksum calculation is done during cell transfers to receive buffer memory.

5.5. Network Environment

- **TCP congestion window** - It is set to the maximum of 65535 bytes to reflect a fatter network pipe based on ATM.
- **MSS** - The default MTU size for IP over ATM is 9180. TCP will negotiate a MSS option of 8K bytes when zero-copy is activated.

5.6. Benchmark Programs

Two programs were used for performance evaluation. Both require small changes to page-align their internal buffers.

- **ttcp** - This is a popular benchmark program for measuring TCP throughput. In our measurement, 16KB write buffer and 64KB read buffer are used. For zero-copy on the transmit side, 144K bytes are allocated and divided into nine circular buffers, each of 16KB in size, This ensures that no COW fault occurs (see section 3.2.4). For each test run, 90000x16K bytes of data are transferred. Both the socket send buffer and receive buffer sizes are set to 64K. The latter effectively sets the size of TCP receive window to 56K[‡‡].

‡‡. This is the maximum TCP window size (without window scale option) rounded down to an integral number of MSS (8K).

- **ftp** - This is one of the most popular network applications. It currently uses a single 8K buffer on both the transmit and the receive sides. Besides aligning the internal buffers, the transmit side is changed to memory-map (mmap(2)) the source file, and use the mapped address directly to write to the network socket. This saves one memory copy from the current implementation, which simply reads data from the file to the internal buffer 8K at a time, then writes the buffer out to the network. The same technique can be used on the receive side, by reading data from the network directly into a memory-mapped file address. Combined with zero-copy, this cuts the number of copy operations from two to zero. But it requires that the target file already exists, and the file size remains unchanged. This is due to the limitation of mmap(2) —it cannot create, grow, or shrink files. For measurement purposes, this technique is not used. Instead, a 1MB file is ftp'ed into /dev/null repeatedly to avoid any file system operations.

5.7. Performance Metric

Comparisons are made between programs running under zero-copy mode and the same programs running under single-copy mode. Three different aspects of performance characteristics are examined -

- **TCP throughput** - The total amount of data transferred is divided by the elapsed time. Numbers are shown in million-bit per second.
- **Code latency** - The on-board micro-second resolution timer is used to compare code latency between zero-copy and single copy. Numbers are shown for operations on 8192 byte buffers (two MMU pages). For example, the number for single-copy is the time it takes to *copyin()/copyout()* 8KB data. Note that, in all test cases on the receive side, each user page has only ONE mapping to it. This is the majority case for the receive side zero-copy. If a user page is being mapped into multiple address spaces, the cost of changing all the mappings will certainly be higher.
- **CPU utilization** - a Sun internally developed tool *statit* similar to *vmstat* is used to gather CPU utilization of the whole system during the test run.

The E-cache controller for SuperSPARC has a special register to track cache misses. This is also a useful performance indicator that is highly correlated with performance metric above. The numbers we show include both the read and write misses.

6. Measurement Results

As we have discussed before, the performance of zero-copy and single-copy depends heavily on the host architecture. Our measurement results on different types of Sun workstations concurs with this. An application's behavior also plays an important role on performance, as we illustrate by using a number of different tests shown below.

6.1. SPARCstation/20 ttcp Test

The following table shows measurement results on SPARCstation/20, both uniprocessor and dual-processor configurations. The other end is connected to a SPARCserver/1000E. Since the latter is a faster machine, performance is bounded by CPU time (<2% idle time) on the SPARCstation/20 end in all test cases. Therefore, throughput numbers can be used as a gauge for overall performance. All numbers shown are obtained by running *ttcp* over 90000x16K bytes of data (see section 5.6).

Table 3. SPARCstation/20 ttcp Test

	Thruput (Mb/s)	Latency (µsec)	CPU (sec) usr+sys	Ecache miss %
UP-T-0	255	115	0+46	3.6
UP-T-1	180	220	1+65	8.2
MP-T-0	220	150	0+55	6.6
MP-T-1	180	220	2+66	9.1
UP-R-0	265	110	0+45	3.3
UP-R-1	175	270	1+66	10
MP-R-0	235	170	0+50	8.2
MP-R-1	185	270	0+64	16
UP-T-0-touch	150	89	32+47	4.9
UP-T-1-touch	135	175	30+56	5.6
UP-R-0-touch	150	60	37+43	8.4
UP-R-1-touch	120	210	32+67	12

T - transmit　　R - receive
0 - zero copy　　1 - single copy
UP - uniprocessor　　MP - multiprocessor

For uniprocessor, zero-copy throughput improves by 42 and 51 percent on transmit and receive sides respectively. The table also exhibits close correlations between latency and E-cache miss rate. Note that due to the write-invalidate bus protocol (see section 5.3), on the receive side, data is completely out of the cache when copy starts. This explains the high E-cache miss rates.

For the MP configuration, expensive software cross calls are used to purge stale TLB entries from remote processors (see section 5.1). On the receive side, it takes four cross-calls to remap two 8K buffers, averaging 15µs ((170-110)/4) per call. The transmit side cost is smaller since two of the four MMU updates are for upgrading user protections from read-only to read-write. Their TLB flushes can be deferred.

6.2. SPARCstation/20-HS11 ttcp Test

HyperSPARC uses a virtually-indexed physically-tagged direct-mapped cache. The cache size is 256KB. When a page is remapped or its protection changed, not only the TLB, but also the cache may need to be flushed. One exception is on the receive side if a page is being remapped into a new address with the same virtual color as the old one, the cache flushing can be deferred. With a MMU page size of 4KB, there are 64 colors (256/4). The chance of avoiding cache flush during a remap is low.

Cache flushing makes page remapping and COW operations more expensive. For the transmit side MP case, zero-copy code takes even longer than single-copy code. But copying seems to cause sufficient negative side effects, possibly due to a small cache size, so that zero-copy still takes less total system time than single-copy.

Many tests here exhibited some idle time. The reason is not clear yet. This is why some throughput numbers are low.

Table 4. SPARCstation/20-HS11 ttcp Test

	Thruput (Mb/s)	Latency (µsec)	CPU (sec) sys+idle
UP-T-0	200	150	58+0
UP-T-1	150	200	75+4
MP-T-0	165	215	67+4
MP-T-1	130	190	76+15
UP-R-0	220	140	52+2
UP-R-1	160	260	72+1
MP-R-0	190	190	64-2
MP-R-1	160	250	77-3

6.3. SPARCserver/1000E/8-way ttcp Test

In this test, throughput numbers no longer accurately reflect actual performance due to idle time caused by slower machine on the other end. Instead, total CPU time should be used as the gauge for performance. We also show per packet processing time, obtained by dividing total CPU time over the number of packets transferred

On the transmit side, zero-copy code takes even longer to run. But the total system time is marginally less than single-copy's. The copy time of 115 µsecs for 8KB implies that the source and destination buffers are relatively warm in the E-cache (see table 2). This is also confirmed by the low E-cache miss rate.

Zero copy performs much better on the receive side, with nearly 30% reduction on total per-packet processing time. Again, source data is cold in the cache, causing the copy time (175µs) to be much higher than that of the transmit side. Moreover, TLB shoot-down is now handled in hardware by bus broadcast. Therefore, the software overhead is less than that of SPARCstation/20 (150 ~ 170µs).

Table 5. SPARCserver/1000E ttcp Test

	Processing time (µs)	Code Latency	CPU (sec) sys	E-cache miss %
T-0	235	140	42	8.8
T-1	265	115	47.5	6.2
R-0	185	85	33	10
R-1	255	175	45	14

6.4. FTP Test

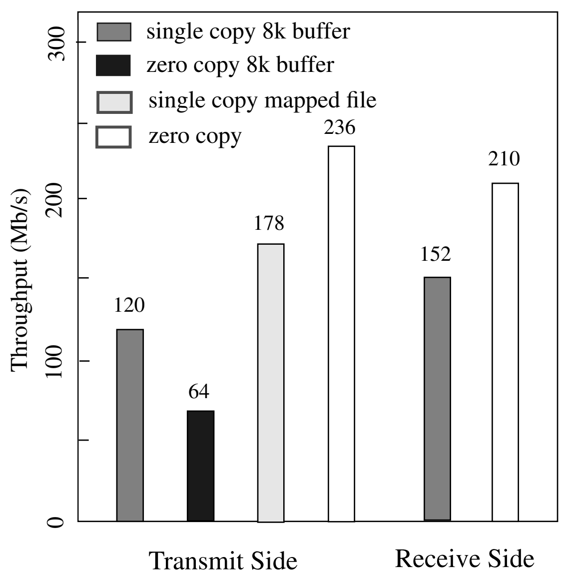
We ran our tests between a SPARCstation/20 and SPARCserver/1000E and collected data on the former. Since the SPARCserver/1000E is faster, CPU time is 100% utilized on the SPARCstation/20.

The transmit throughput nearly doubles compared to the current two-copy scheme − first from the source file page to a user buffer, then from the user buffer to a network buffer. The saving of one memory copy by using memory mapped file address accounts for about half of the performance gain.

Normally the kernel will not enable zero-copy unless the total size of "working" write buffers exceeds certain threshold. This is to avoid excessive COW faults described before. To illustrate how much impact excessive COW faults may have on zero-copy performance, the internal COW-fault-avoidance threshold is reduced to only 8K, and the 8K FTP write buffer is used. The result is a almost 50% throughput drop.

The receive side gets nearly 40% throughput boost. In a real application where incoming data is directed to a file, data access penalty due to a cold cache may cut into the gain (see next section). The best arrangement is to connect the zero-copy network to another zero-copy I/O path, such as memory mapped files. In that case, performance gain similar to the magnitude seen on the transmit side may be realized on the receive side.

Figure 2: FTP Test on SPARCstation/20-UP



6.5. Data Touching Penalty

All the test cases so far move data from one node to another without manipulation. In reality, applications often write into the data before transmitting, and read from the data after receiving. Touching the data may cause zero-copy's performance edge over single-copy to diminish. Specifically, a warmer cache after data is touched on the transmit side will speed up single-copy, and a colder cache after zero-copy on the receive side will slow down applications accessing the data. To measure the impact, ttcp is modified to write into every word of the write buffer before transmit, and to read from every word of the data after it is received. The result is shown in the last four rows of table 3.

On the transmit side, the copy latency and total system time drop noticeably, apparently due to a warmer data cache. On the receive side, zero-copy costs ttcp a higher user time than single-copy due to a colder data cache. This is also confirmed by its higher cache miss rate than the case without touching the data. On both sides, zero-copy is still a performance win overall.

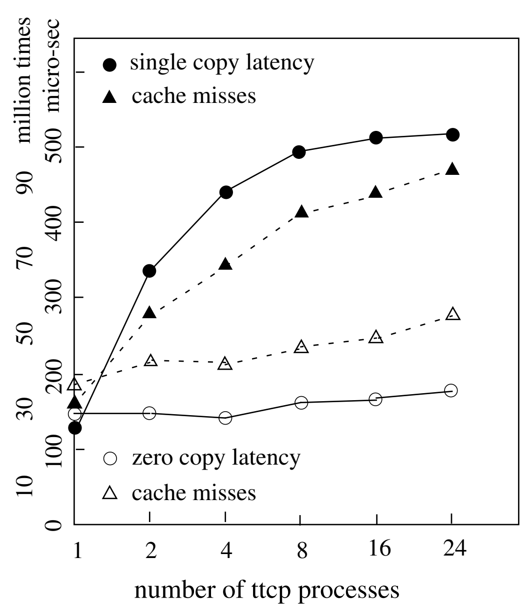
6.6. Loaded System Test

It is imperative to study how zero-copy performs under load compared to single-copy. This is especially important for servers where there are often multiple tasks running concurrently. On one hand, increasing VM lock contention may hurt zero-copy performance. On the other hand, thread migration caused by higher CPU contention, plus excessive copy operations may greatly reduce the effectiveness of the hardware cache, and affect single-copy performance.

We ran multiple copies of ttcp simultaneously on SPARCserver/1000E to simulate a loaded system. Zero-copy shows little sign of VM lock contentions, and turns out to scale well with the number of running processes. On the receive side, single-copy's performance doesn't change much, possibly due to a cold cache that can't get any worse. Nevertheless, on the transmit side the warmer data cache that helps single-copy performance with only one ttcp running becomes a liability when multiple ttcp processes contend and migrate among different CPUs. This is especially damaging on SPARCserver/1000E, where an access to a remote E-cache is slower than main memory.

The following figure demonstrates this phenomenon. When two copies of ttcp are run, copy time of 8K bytes more than doubles to 325µs from 115µs. This is also reflected by the number of cache misses. With more copies of ttcp running, single copy latencies increase steadily until saturated around 500µs, whereas both the cache misses and latency curves for zero-copy stay flat.

Figure 3: Loaded System Test on SS1000E



7. Conclusions and Future Work

This paper presents an efficient zero-copy implementation for network I/O. It discussed trade-offs among several zero-copy schemes, and contends that our design based on virtual memory page remapping and copy-on-write techniques require the least amount of changes to the existing system and applications, while still offer significant performance gain.

Performance improvements vary somewhat, depending on how efficiently a host's memory cache system can copy data, versus how much overhead MMU operations require. Furthermore, zero-copy works best with other copyless I/O paths, such as memory mapped file, where data is moved to or from disks without CPU copy.

On the transmit side, COW faults are expensive, so is setting up a COW protection on a user buffer, and tearing it down later. Applications need to use a chain of buffers of some sufficiently large size in order to avoid COW faults. Without end-to-end acknowledgments, they don't know when the transport is finished with a buffer. Therefore, COW protections are necessary to ensure data integrity. Nevertheless, if zero-copy is combined with asynchronous I/O facility, so that applications get notified when a buffer is released by the transport and can be safely reused, COW protection won't be necessary. The saving is significant as demonstrated by the following test.

Table 6. Transmit side zero-copy w/o copy-on-write protections

	Processing time (µs)	Latency (µs)	CPU sys (sec)
SS20-UP-0-COW	255	115	46
SS20-UP-w/o COW	190	72	34
SS20-UP-1	360	220	65
SS1000E-0-COW	235	140	42
SS1000E-w/o COW	135	65	24
SS1000E-1	265	115	47.5

With network adaptors capable of calculating TCP checksum, normally networking software no longer needs to have access to user data. This presents another opportunity for further performance improvement. Physical data pages may simply flow through the kernel domain without being mapped. On the transmit side, it saves all the kernel mapping cache work. On the receive side, unmapped and unnamed pages may simply be mapped into user address space and named appropriately. The amount of page remapping and renaming work will be only half of what is required now. The potential benefit is even greater for NFS***, as physical pages carrying client data can be renamed and written out to disk without ever being mapped.

Acknowledgments

Neal Nuckolls first proposed the project. Erik Nordmark offered many great insights and suggestions. Bruce Curtis implemented the hardware checksum support and gave valuable critics. Denny Gentry and Jerry Chen provided the ATM hardware and device driver support.

I'd also like to thank Bill Shannon and the VM group for reviewing the VM code, Mike Tracy for scrutinizing the STREAMS code, Wolfgang Thaler for proof reading the paper, and Helen Vanderberg for correcting my writing. Special thanks to Anil Shivalingiah for introducing me into the VM system many years ago.

Most of all, I thank my dear wife Wendy for sparing my many, many weekends at work.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[2] D. R. Cheriton. "The V distributed system," *Communications of the ACM*, vol.31, no.3, March 1988.

[3] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine,* 23-29, June 1989.

[4] E. Cooper, P. Steenkiste, R. Sansom, and B. Zill. "Protocol Implementation on the Nectar Communication Processor," *Proceedings of SIGCOMM '90 Conference on Comm. Architectures, Protocols and Applications,* Aug. 1994.

[5] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. "Afterburner - A network-independent card provides architectural support for high-performance protocols," *IEEE Network*, July 1993.

***. Due to a buffer alignment issue described in Section 3.2.4, zero-copy is currently not supported in NFS.

[6] P. Druschel and L Peterson. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles,* Dec. 1993.

[7] P. Druschel, L. Peterson, and B. Davie. "Experiences with a High-Speed Network Adaptor: A Software Perspective," *Proceedings of SIGCOMM'94 Conference on Comm. Architectures, Protocols and Applications,* Aug. 1994.

[8] R. Fitzgerald and R. F. Rashid. "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM Transactions on Computer Systems*, 4.2, May 1986.

[9] R. Gingell, J. Moran and W. Shannon. "Virtual Memory Architecture in SunOS," *Proceedings of the USENIX Conference*, May 1987.

[10] J. Kay and J. Pasquale. "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000," *Proceedings of the Winter USENIX Conference,* Jan. 1993.

[11] K. Kleinpaste, P. Steenkiste, and B. Zill. "Software Support for Outboard Buffering and Checksumming," *Proceedings of SIGCOMM'95 Conference on Comm. Architectures, Protocols and Applications,* Aug. 1995.

[12] J. Moran. "SunOS Virtual Memory Implementation", *Proceedings of EUUG Conference.* London England, Spring 1988.

[13] J. Smith and C. Brendan S. Traw. "Giving Applications Access to Gb/s Networking," *IEEE Network,* July 1993.

[14] P. Steenkiste. "A Systematic Approach to Host Interface Design for High-Speed Networks," *IEEE Computer,* March 1994.

[15] B. Traw. "Applying Architectural Parallelism to High Performance Network Subsystems," *Ph.D. Dissertation, University of Pennsylvania*, 1995.

[16] S.-Y. Tzou and D. P. Anderson. "The Performance of Message-passing using Restricted Virtual Memory Remapping," *Software - Practice and Experience*" vol. 21, March 1991.

[17] A. Wolman, G. Voelker, and C. Thekkath. "Latency Analysis of TCP on an ATM Network," *Proceedings of the Winter USENIX Conference,* Jan. 1994.

Author Information

H.K. Jerry Chu is a Staff Engineer with the Internet Engineering Group at Sun Microsystems. He has been with Sun since 1987, working in many different areas of the operating system kernel, including file systems, VM, low-level porting and system bring-up. His latest venture is in the networking world.

Jerry received his B.S. in Mathematics from National Taiwan University in 1979. While in graduate school, he gave up his youthful dream of becoming a mathematician, and found consolation in computer programming. He received an M.S. in Statistics and an M.S. in Computer Science from Stanford University in 1982 and 1984 respectively. He can be reached electronically at *jerry.chu@Eng.Sun.COM*.