Revisiting Software Zero-Copy for Web-caching Applications with Twin Memory Allocation*

Xiang Song† ‡, Jicheng Shi† ‡, Haibo Chen†, Binyu Zang† ‡
†Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
‡Software School, Fudan University

ABSTRACT

A key concern with zero copy is that the data to be sent out might be mutated by applications. In this paper, focusing specially on web-caching application, we observe that in most cases the data to be sent out is not supposed to be mutated by applications, while the metadata around it does get mutated. Based on this observation, we propose a lightweight software zero-copy mechanism that uses a twin memory allocator to allocate spaces for zero-copying data, and ensures such data is unchanged before being sent out with a lightweight data protection mechanism. The only change required to an application is to allocate zero-copying data through a specific ZCopy memory allocator. To demonstrate the effectiveness of ZCopy, we have designed and implemented a prototype based on Linux and ported two applications with very little effort. Experiments with Memcached and Varnish shows that show that ZCopy can achieve up to 41% performance improvement over the vanilla Linux with less CPU consumption.

1 INTRODUCTION

Many network-intensive applications can easily be limited by the speed of network I/O processing. Other than the physical limitation of networking devices, the performance of networking applications are also constrained by the efficiency of network I/O sub-systems, in which data copying is one of the key limiting factors. Usually, during network protocol processing, the operating system kernel has to copy data from user space to a kernel buffer and then sends the kernel buffer to the network device.

Though there has been extensive research on avoiding the data copying, prior systems are still not easily adoptable for many applications on commodity operating systems with commodity networking devices. One approach is bypassing the operating system with Remote DMA. However, these require special and expensive hardware (e.g., Infiniband [2] and Myrinet [10]) and most commodity networking devices have not been built with such support. Several previous software zero copy mechanisms, such as fbufs [7] and IO-Lite [11] are designed for a micro-kernel and require special data management and accessing methods across protection domains. Container shipping [12] supports zero-copy on UNIX platforms, but requires data being aggregated in a scatter-gather manner and additional system-call interfaces. Approaches [5, 6] using on-demand memory mapping and copy-on-write mechanism are limited by the protection granularity (e.g., page size) and the corresponding alignment requirement, thus may face the false sharing problem that protects unwanted data. This may cause notable performance overhead for irregular (e.g., unaligned) data chunks. Modern operating systems also have several mechanisms to support zero copy, such as sendfile [14] and splice [9]. However, such mechanisms require zero-copying data to be treated as files, which is not feasible in many applications that need to mutate the data to be sent out.

The key issue in supporting software zero-copy is that the zero-copying data might be mutated when being sent out. This is because when a user application invokes a data sending system call (e.g., sendmsg and write), it assumes that the data has been sent out when the system call returns. However, when such system calls return, the data might have not been moved into the networking devices. If the kernel does not copy the data from the user buffer to a kernel buffer, any changes on the data from applications may be sent out, which violates the semantics of such system calls.

Intuitively it should be the case that the data will normally not be mutated. However, focusing specifically on web-caching applications, we observe that, in most cases, the data to be sent out is not supposed to be mutated by applications. However, the data around it, especially the metadata corresponding to it, does get mutated. Some metadata (e.g., the data expire time in Memcached [8]) is usually co-located around the data to be sent. Due to lacking of application semantics, operating

*We thank our shepherd Alexandra Fedorova the anonymous reviewers for their insightful comments. This work was funded by China National Natural Science Foundation under grant numbered 61003002, a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100. Xiang Song was also funded by Fudan University's outstanding doctoral research funding schemes 2011.

systems cannot simply zero-copy a page with specific network data packets as that page holding the network data can be modified by applications.

Based on the above observation, we revisit the software zero-copy mechanism for web-caching applications. The basic idea is using a second (twin) memory allocator to allocate and aggregate data that are likely to be zero-copied, and providing a lightweight memory protection mechanism in case such data does get modified. Hence, the zero-copying data can be isolated from other application data, thus can be aggregated together to allow kernel to use traditional page-level protection. This minimizes unnecessary write protection faults due to false sharing. To support software zero copy, an in-kernel proxy is added into the UDP and TCP processing paths to distinguish the zero-copy data with the others. A write protection module is also added to handle rare cases where the data that is supposed to be zero-copied have really been mutated. In such a case, the data will be copied to ensure program correctness.

We have implemented a prototype based on Linux 2.6.38. The prototype of ZCopy is very lightweight and adds around 735 lines of code (LOCs) to Linux kernel and adds 20 LOCs to streamflow [13]. It consists of a specific user-level memory allocator ZC_alloc based on streamflow. A 200 LOCs user-level library is implemented to support cooperation between the ZCopy kernel and the ZC_alloc to provide memory protection for zero-copying data.

The porting effort required to run web-caching applications on ZCopy using zero-copy mechanism is also quite small. Providing zero-copy support to Memcached [8], a widely-used key-value based memory caching server, requires only 10 LOCs changes. Running Varnish [4] server also only requires 3 LOCs modification. The only change required is simply replacing the memory allocator for zero-copying data with the one provided by the ZC_alloc.

To measure the effectiveness of ZCopy, we conducted several application performance measurements using Memcached and Varnish web caching system. Performance results show that ZCopy brings modest improvement over vanilla Linux. ZCopy improves the throughput of Memcached over vanilla Linux up to 41.1% and 40.8% for UDP and TCP processing when the value size is larger than 256 bytes. The performance speedup of Varnish ranges from 0.7% to 7.9% for data size ranging from 2 KBytes to 8 KBytes.

2 Observation

To gain insight into how network data might be mutated, we make a case study on Memcached. Figure 1 shows the basic storing item structure of Memcached to store key/value pairs. The key/value data is stored at the end of *stritem*, while the metadata is stored from the beginning of it. Each time Memcached receives a request and find a corresponding key/value pair, the refcount of the corresponding item will be increased in function *do_item_get* (As shown in Figure 2). If we write protect the key/value pair, we need also write protect the metadata around it. Hence, there will be a lot of unnecessary protection faults due to false sharing.

The example indicates that for some networking applications, the network I/O data to be sent out is not supposed to be mutated. However, the data around it, especially the metadata corresponding to it does get mutated. Hence, naively write-protecting the networking data may also protect the metadata allocated within the same page, resulting in false protection.

```
typedef struct _stritem {
1   struct _stritem *next;
2   ...
3   uint8_t          nkey;     /* key length */
4   void * end[];              /* struct {  keyn
                                            suffix
                                            data  } */
} item;
```

Figure 1: *Memcached storing item structure.*

```
1 item *do_item_get(const char *key, const size_t nkey) {
2       item *it = assoc_find(key, nkey);
3       ...
4       if (it != NULL) {
5               it->refcount++;
6       ...
7 }
```

Figure 2: *Code piece of function do_item_get.*

3 Design and Approaches

This section first presents an overview of ZCopy and then illustrates the approaches to supporting efficient zero-copy mechanism.

3.1 ZCopy Overview

It is quite intuitive to let applications to designate which data should be zero-copied. When such data is being sent out, ZCopy will zero-copy it while processing other data through the normal path. However, it has to deal with the following issues: 1) it should retain the existing memory accessing manner for user applications; 2) it should conform to existing system calls to avoid adding any new interfaces; and 3) it should provide proper protection over the data to be sent out to conform to the semantics of existing network sending system calls.

In ZCopy, we introduce a twin memory allocator to separately allocate data according to application semantics and aggregate several zero-copying memory blocks into the same memory chunks. Hence, the data to be

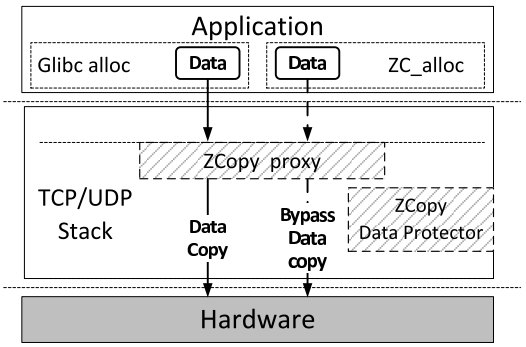

Figure 3: *An overview of architecture of ZCopy*

protected can be separated from other application data. Figure 3 shows the general architecture of ZCopy. The application running on ZCopy can use the original memory allocator (e.g., glibc) to allocate memory for normal data or use the twin memory allocator named ZC_alloc to allocate memory for zero-copying data. A ZCopy proxy is added to the UDP and the TCP package processing path to distinguish the network data that will be zero-copied from others. If the data is allocated from ZC_alloc, ZCopy will bypass the data copy path. Otherwise, ZCopy will handle the data as usual. The proxy also cooperates with the ZCopy data protection module to provide basic write protection on the zero-copying data.

3.2 Supporting Zero-copy

3.2.1 Isolating Zero-copying Data with Twin Memory Allocator

To isolating zero-copying data from other data, ZCopy provides a twin memory allocator along with the original one to allocate memory for network data that is guaranteed to be insulated from other data allocated from a generic memory allocator (e.g., glibc). Restricted by the minimal memory protection granularity of a page size and the following address alignment requirement, ZC_alloc has to pay special attention to small memory blocks (e.g., block size small than 1024 bytes). A naive way to handle this is to allocate one page for each request. However, this may waste a lot of memory. ZC_alloc uses an aggressive way by aggregating memory blocks with similar sizes into the same basic memory unit, namely the pageblock. A pageblock is treated as a basic protection chunk and usually consists of several pages (16 pages by default). It is write protected only when it is full of zero-copying data. As ZC_alloc aggregates zero-copying data together to provide memory protection, it minimizes the amount of wasted memory (e.g., by aggregating small objects smaller than 1 page size into a default pageblock, the maximum amount of memory wasted is less than 1 page, which is less than 6.25%).

If the allocation request is for a large data block, ZC_alloc directly allocates a memory chunk rounded from the requesting size. A threshold (4096 bytes by default) is set in ZC_alloc to decide whether a request is for the large data block. This threshold can be tuned by the programmer if needed.

The twin memory allocator is especially friendly to the reusable data. Once a data block is allocated it will be sent out to network multiple times before it is modified or freed. One representative usage scenario is allocating value data for Memcached. Memcached server caches a lot of key/value pairs in memory to serve quick key/value queries. Every time the server receives a request containing a key, it will respond with the value corresponding to that key. For the perspective of long execution, the key/value pairs are not expected to be modified or freed. Hence, we can zero-copy the value during data transferring without worrying about the modification to such data in most cases.

3.2.2 Zero-copying Network I/O Data

ZCopy supports two common network protocols: UDP and TCP. We add a proxy in UDP and TCP's package processing paths to distinguish the network data that will be zero-copied and others. At the very beginning, ZCopy will first check whether current process wants to use zero-copy mechanism or not. If so, it will check whether there are any memory blocks that need to be write protected. The ZCopy data protection module is invoked if write protection is needed.

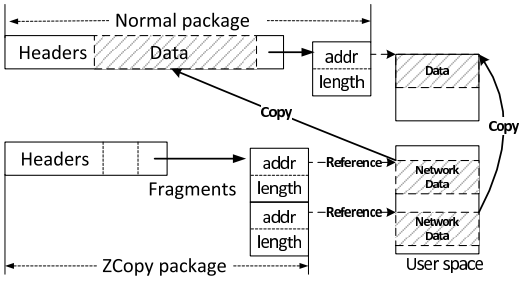

Figure 4: *The structure of normal package and ZCopy package*

ZCopy handles zero-copying data at the time when the network data is organized into a network package. Figure 4 shows the structure of normal network package and the ZCopy package. In normal cases (shown in the top half of Figure 4), a network package consists of several protocol headers followed by network data. The network data can be organized as a single data buffer or a list of data buffers. The data is copied from user address space into the package in order. If the package buffer is not large enough to hold all network data, the kernel will allocate new empty pages to hold the rest of the data and attaches them into the package's page fragment list. Each entry in the list contains the starting address of the data

and its length. When the package is passed to the NIC driver, the driver will first transfer the package content and the fragments to the NIC hardware through the DMA engine.

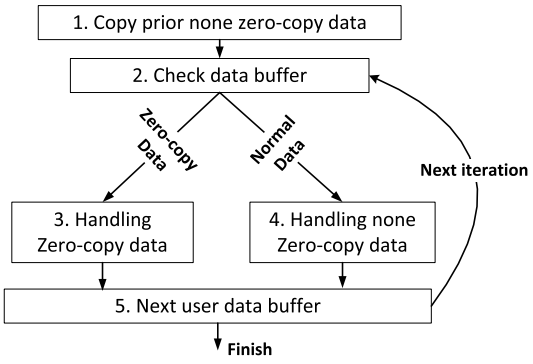

Figure 5: *Zero-copy in UDP package processing*

ZCopy treats zero-copying data differently from normal data. Each pageblock is identified by a magic string, thus the zero-copy data buffers can be distinguished with others. We use the UDP package processing as an example to illustrate the process of handling zero-copying data. Figure 5 shows the UDP package processing path in ZCopy and the bottom half of Figure 4 shows the structure of a ZCopy package. ZCopy first scans the user data buffer lists to copy all prior normal data into the network package buffer including the protocol headers (step 1). Then, it iteratively processes the following user buffers by handling zero-copying data and normal data separately (step 2-5). It will check the pageblock magic string to discover zero-copying user buffers. For zero-copying data (step 3), it first gets the starting address and the length of the data buffer. It then finds all pages covered by the data buffer and finally organizes the pages in the form of fragments and adds them into the package's page fragment list. For normal data (step 4), it allocates new empty pages and copies the buffer content into them. ZCopy finally organizes the pages in the form of fragments and adds them into the package's page fragment list. The package will be passed into the lower level of the network stack.

One optimization to the ZCopy proxy is to treat read-only data buffers as zero-copying buffers, though they are not allocated using ZC_alloc. This can simply be done by feeding the offset and length in the fragment list.

3.2.3 Protection of Zero-copying Data

ZCopy must provide a protection mechanism to the zero-copying data in case it is mutated when the data is sent out. To do this, ZCopy adds a simple data protection module into the native memory management system.

Based on the page-level protection granularity in kernel, small data blocks allocated from ZC_alloc are batched in a group and are treated as a whole for write protection. The minimal protection unit is one pageblock. When a pageblock is full, ZC_alloc will request the kernel to protect it. To avoid the cost of context switches between user space and kernel space and possible false protection problem caused by early write protection, ZCopy batches the requests from ZC_alloc to delay the protection of the pageblock until the system enters the network package processing path. The protection is done by walking the page table of the target range and changing the protection bit of the corresponding page table entries. When the pageblock is not full, data allocated from ZC_alloc are still sent through normal path without being zero-copied.

ZCopy tries to protect zero-copying data blocks in an aggressive way. ZCopy does not remove the write protection of the data block even if the data block is completely sent out by the hardware. The removal of the write protection is triggered only when a write operation is trapped by the kernel. At that time, the reference count of the page corresponding to the faulting address is first checked. If the count is larger than one, a copy-on-write mechanism is used to protect the network data from being modified. Otherwise, the changing request should come from the application itself and we simply remove the write protection. Note that, the basic protection unit is a pageblock, any write to a write protected pageblock will cause all the data blocks belong to the pageblock lose the write protection. However, we do not expect this happens frequently as mutation on zero-copying data is rare.

4 Experimental Results

All experiments were conducted on an Intel machine with 2 1.87 Ghz Six-Core Intel Xeon E7 chips running Debian GNU/Linux 6.0 with the kernel version 2.6.38. The NIC used is an Intel 82576 Gigabit Network Controller. We use another Intel machine with the same hardware and software configuration as the client machine. To minimize the interaction between different cores of a multi-core system (e.g., cache trashing), experiments were conducted using only one CPU core.

We use two widely-used web-caching applications, Memcached 1.4.5 [8] and Varnish 3.0.0 [4] to demonstrate the performance improvements. All applications in the experiments use the ZC_alloc to allocate memory for network data to eliminate the effect of using different memory allocators.

4.1 Memcached

Memcached [8] caches multiple key/value pairs in memory. Each time it receives a request containing a key, it will respond with the corresponding value. From a long run's perspective, the key/value pairs are not expected to

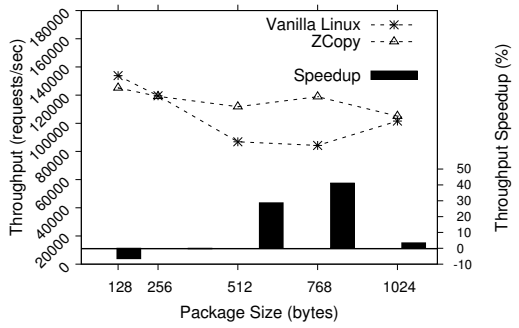

Figure 6: *The throughput of Memcached in ZCopy and vanilla Linux with UDP and the speedup of ZCopy.*

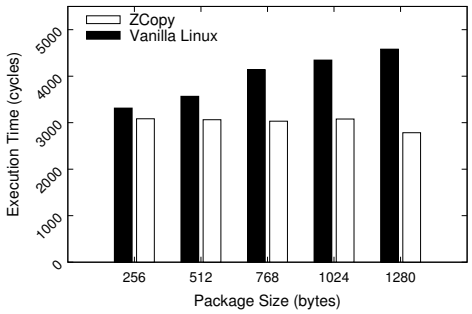

Figure 7: *The time spent on UDP package processing for Memcached in ZCopy and vanilla Linux.*

be modified or freed. However, the metadata (e.g., the data expire time, the item links) along with the cached pairs may change. We modify Memcached to allocate memory for the values from ZC_alloc. This takes only 10 lines of modification to the original Memcached.

We use the memaslap testsuite form the libmemcached library [3] as the client of Memcached. The client first warms up Memcached with a user-defined number of key/value pairs and then randomly issues get and set operations through several concurrent connections.

UDP: Figure 6 shows the average throughput of Memcached in ZCopy and vanilla Linux. The Memcached is warmed up with ten thousand key/value pairs. The memaslap client is configured to issue pure get operations through 36 concurrent connections from 12 threads using the UDP protocol. We adjust the number of worker threads of Memcached to achieve the best performance. The CPU usage in all cases is above 99%. Vanilla Linux performs slightly better when the value size is smaller than 256 bytes. However, when the value size reaches 512 bytes, ZCopy starts to outperform vanilla Linux. In 512 bytes cases, ZCopy has a 28.7% performance improvement. When the value size is 768 bytes, the performance improvement increases to 41.1%. For the case where the value size is 1024 bytes, ZCopy and vanilla Linux has nearly the same throughput as the network reaches its hardware limitation.

The performance improvement comes from two parts: 1) minimized data copying and 2) reduced cache trashing. Figure 7 compares the time spent on UDP package processing in ZCopy and vanilla Linux. In ZCopy, the package processing time is around 3000 cycles in all cases. However, in vanilla Linux, the time increases along with the package size and reaches 4400 cycles in 1024 bytes cases. Table 1 shows the L2 cache miss rate of Memcached in Linux and ZCopy. ZCopy reduces more than 10% L2 cache misses in UDP cases. The hottest function *copy_user_generic_string* in Linux disappears in ZCopy. Another reason for such notable performance improvement in the 512 and 768 cases is that the shorter package sending time in ZCopy causes the NIC interrupt handler switch frequently to the polling mode which is more effective than the interrupt mode in heavy network stress. However, in vanilla Linux, the network status triggers less frequent switches to the NIC polling mode.

L2 Cache Miss Rate (1 miss/K cycles)			
	512 bytes	768 bytes	1024 bytes
UDP Linux	4.89	5.17	6.11
UDP ZCopy	4.17	4.57	4.73
TCP Linux	8.08	9.06	10.86
TCP ZCopy	7.73	8.22	9.46

Table 1: *The L2 cache miss rate in vanilla Linux and ZCopy in 256 byte, 768 byte and 1024 byte cases.*

TCP: Figure 8 shows the average throughput of Memcached in ZCopy and vanilla Linux and the performance speed of ZCopy over vanilla Linux. We use the same evaluation method used in the UDP experiments. For each TCP connection, we only issues a signle request and then close it. Vanilla Linux performs better when the value size is smaller than 256 bytes. However, when the value size reaches 512 bytes, ZCopy starts to outperform the vanilla Linux by 40.8%. When the value size is with 1024 bytes, ZCopy outperforms vanilla Linux by 30.8%. The performance of Memcached reaches the hardware limits when the value size is of 2048 bytes.

As in UDP, the performance improvement comes from copy avoidance and reduced cache trashing. As the code for TCP package processing and data sending is mixed together, we measure the time spent on the *tcp_sendmsg* instead of TCP package processing time. Figure 9 shows the profiling results. From the figure we can see that ZCopy does reduce the time spent on *tcp_sendmsg* in all cases. Table 1 shows the L2 cache miss rate of Memcached in Linux and ZCopy. ZCopy reduces 10.2% L2 cache misses in 768 byte cases and 14.8% L2 cache misses in 1024 byte cases.

4.2 Varnish

Varnish [4] is an open-source web application accelerator. It caches web content into memory objects and re-

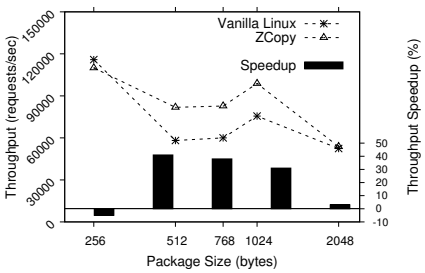

Figure 8: *The throughput of Memcached in ZCopy and vanilla Linux with TCP and the speedup of ZCopy.*

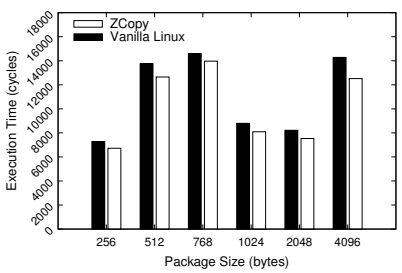

Figure 9: *The time spent on function tcp_sendmsg for Memcached in ZCopy and vanilla Linux.*

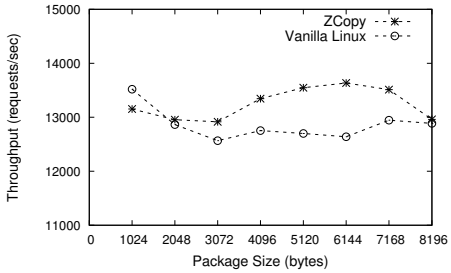

Figure 10: *The throughput of varnish server in ZCopy and vanilla Linux.*

turns web objects according to the network request. We modify Varnish to allocate object memory from ZC_alloc with 3 LOCs changes.

We test Varnish using ab (apache benchmark) from Apache with the web page sizes ranging from 1 KBytes to 8 KBytes (the average individual response size ranges from 3 KBytes to 15 KBytes [1].) Figure 10 compares the performance of ZCopy and vanilla Linux. The varnish server saturates the CPU on both ZCopy and vanilla Linux. Vanilla Linux performs slightly better with small web page sizes (1 KBytes). However, when the web page size increases, ZCopy starts to outperform Linux. The performance improvement reaches 7.8% when the web page size increases to 6 KBytes. Both configurations reach networking limitation when the web page size increases to 8 KBytes. The reason that the improvement is much less than Memcached is that the single request processing time in Varnish is much longer than that in Memcached, which thus amortize the improvements of ZCopy.

	CPU cycles
getpid	1149.9
ZCopy write protection fault	2802.5
native page fault	6247.4

Table 2: *The execution time of invoking getpid system call, triggering ZCopy write protection fault and triggering native page fault.*

4.3 ZCopy Primitive

Overhead of Write Protection We also evaluate the cost of triggering write protection faults for zero-copied data. Table 2 shows the execution time of invoking the getpid system call, triggering ZCopy write protection fault and triggering traditional page fault respectively. The cost of triggering a ZCopy write protection fault is much smaller than triggering a native page fault. This is because usually ZCopy only removes the write protection of the faulting address from the page table, which is much less expensive.

5 Conclusion and Future Work

This paper revisited the existing software zero-copy mechanism and presented a new zero copy system named ZCopy, which was based on the observation that the metadata around the network data will usually get mutated. Experiments with two applications on an Intel machine show that ZCopy outperforms vanilla Linux for sending a relative large network data package.

In our future work, we plan to extend our work in two directions. First, though we focus specially on web-caching applications in this paper, ZCopy places little constraints on applications and is applicable to other networking applications. We plan to study and evaluate the performance benefit of ZCopy on other network-intensive applications. Second, ZCopy was evaluated using a single core. We plan to extend the ZCopy to efficiently run on multicore machines.

References

[1] Average web response size. http://www.httparchive.org/.

[2] Infiniband. http://www.infinibandta.org/.

[3] LibMemcached. http://libmemcached.org/.

[4] Varnish web cache system. https://www.varnish-cache.org/.

[5] J.C. Brustoloni and P. Steenkiste. Effects of buffering semantics on i/o performance. In *Proc. OSDI*, 1996.

[6] Jerry Chu. Zero-copy tcp in solaris. In *Proc. Usenix ATC*, 1996.

[7] P. Druschel and L.L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. SOSP*, pages 189–202. ACM, 1993.

[8] R. LERNER. Memcached integration in rails. *Linux Journal*, 2009.

[9] L. McVoy. The splice i/o model, 1998.

[10] myricom. Myrinet. http://www.myricom.com/scs/myrinet/overview/.

[11] V.S. Pai, P. Druschel, and W. Zwaenepoel. Io-lite: a unified i/o buffering and caching system. *ACM TOCS*, 18(1):37–66, 2000.

[12] J. Pasquale, E. Anderson, and P.K. Muller. Container shipping: operating system support for i/o-intensive applications. *Computer*, 27(3):84–93, 1994.

[13] S. Schneider, C.D. Antonopoulos, and D.S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proc. ISMM*, pages 84–94, 2006.

[14] D. Stancevic. Zero copy i: user-mode perspective. *Linux Journal*, 2003(105):3, 2003.