Memory Management Strategies for Data Serving with RDMA

Dennis Dalessandro
Ohio Supercomputer Center
1 South Limestone St., Suite 310
Springfield, OH 45502
dennis@osc.edu

Pete Wyckoff
Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Abstract

Using remote direct memory access (RDMA) to ship data is becoming a very popular technique in network architectures. As these networks are adopted by the broader computing market, new challenges arise in transitioning existing code to use RDMA APIs. One particular class of applications that map poorly to RDMA are those that act as servers of file data. In order to access file data and send it over the network, an application must copy it to user-space buffers, and the operating system must register those buffers with the network adapter. Ordinary sockets-based networks can achieve higher performance by using the "sendfile" mechanism to avoid copying file data into user-space buffers.

In this work we revisit time-honored approaches to sending file data, but adapted to RDMA networks. In particular, both pipelining and sendfile can be used, albeit with modifications to handle memory registration issues. However, memory registration is not well-integrated in current operating systems, leading to difficulties in adapting the sendfile mechanism. These two techniques make it feasible to create RDMA-based applications that serve file data and still maintain a high level of performance.

1 Introduction

High-performance computing (HPC) has continuously pushed the bounds of networking, and has ushered in the adoption of faster and more capable interconnects. While common in HPC today, 10 Gigabit networks are only beginning to appear in the mainstream data center. This push into non-HPC environments brings with it new challenges in adapting existing communication models to use newer and faster networks.

One of the core network technologies used to achieve high performance is Remote Direct Memory Access (RDMA). Popular networks such as Infiniband, iWARP [12], and Quadrics take advantage of RDMA. Overall, the use of RDMA in applications has to date been limited to specialized domains. While part of the reason for this may be due to availability and marketing, it may also be that the difficulty in programming for RDMA devices is slowing the uptake of the technology. Years of work have gone into building up a massive infrastructure around sockets-based networking, all of which must be re-addressed in the context of RDMA devices. We address in this work the issue of handling memory registrations when sending file data.

Currently the most popular use of RDMA-based adapters is in HPC, where the dominant programming model is message passing. Application programmers explicitly send and receive data but do not manage memory registrations directly. In the library of an MPI implementation, the most effective scheme to improve performance is to cache the registrations of recently used memory regions. This works, but has numerous outstanding problems for applications that use some of the more "interesting" features of virtual memory in an operating system (OS), such as `fork` and `mmap` [15]. MPI does include an I/O component but does not provide an API for the server side of an implementation.

Beyond MPI, there are a few specialized applications that can take advantage of RDMA networks. Distributed file systems, including NFS-RDMA and PVFS, use RDMA to great advantage. These efforts and others have complex memory management and flow control schemes that address the registration issue. There is another, more general, class of applications that serve data in some fashion that could use RDMA networks. These include servers for FTP, HTTP, CIFS, DAV, torrent, and many more popular Internet-scale protocols. However, it is difficult to manage memory registration in a way that achieves good performance and scales well with the number of clients or requests.

2 The File Serving Problem

A key feature of RDMA is "zero copy," the ability for the network device to move data to and from user memory without involvement of the CPU or OS. This saves CPU cycles to copy the data, and more importantly, reduces traffic on the memory bus.

One of the major complicating factors in using RDMA networks is the need to register memory. Before buffers can be used, they must be pinned down in the kernel, and translated to physical addresses to hand to the network card. Registration cost increases with the size of the buffer and is significant: for a 1 MB message, Infiniband 4X SDR achieves a transfer rate of roughly 900 MB/s, but when registration is included in the timing loop, this drops to around 600 MB/s [15].

It may seem that since the user does not have to explicitly register memory with Quadrics, that the problem would be solved. However, memory registration still happens implicitly when the adapter needs another region. If a large number of clients are being handled simultaneously, it could result in memory thrashing, and hence make explicit memory registration more attractive.

Zero-copy transfers work well when data is generated by the application and thus already resident in user buffers. The problem occurs when needing to transfer data that conceptually lives in the operating system. This happens, for instance, when sending data from the local file system, as a web server or FTP server would do. The act of getting the file into user application memory introduces possibly multiple copies of data, and while the network may avoid any further copies in the transfer between hosts, there is nothing to be done about the data copy from the operating system.

We consider, in this work, how to ship file contents across RDMA networks. All mechanisms are complicated by the need to register memory. Pipelining is one likely approach, but more promising is the "sendfile" mechanism of Linux and BSD operating systems, which allows sending a file over a TCP/IP socket without copying the file's contents to user space. The kernel simply reads from the file when generating outgoing packets. Unfortunately, RDMA connections can not be represented as file descriptors to streaming sockets, and thus can not currently take advantage of the sendfile operation.

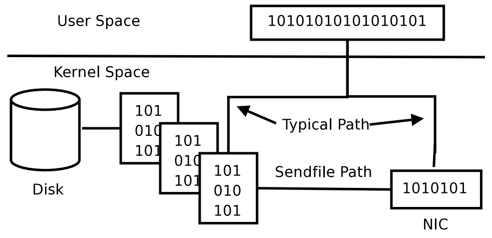

Figure 1. Data paths for file serving.

2.1 Data Path

In applications that involve a client-server communication paradigm, the overwhelming majority store and retrieve persistent data in the local file system. This is obvious in the case of FTP or CIFS whose goal is to access files on a remote server, but also true in the case of serving static content via HTTP, DAV, or torrent. Interactions with the local file system through the operating system are thus required.

Figure 1 shows the typical path for sending data from a file across a network. First, the application initiates a read operation that brings the data from disk into the kernel's page cache, which is then copied to the destination user space buffers. Next, the application writes the data to the network, causing another data copy back to kernel socket buffers. This latter copy is avoided by using the *OS bypass* feature of RDMA networks. Future accesses to the same file will avoid a disk access and just copy data from the kernel page cache to user buffers.

Figure 1 also shows the data path that results when using the *sendfile* approach. Instead of reading the file data into a user space buffer then writing it back to kernel socket buffers, the kernel directly builds outgoing packets using data from its page cache. Avoiding two memory copies is a great improvement and has led to the widespread adoption of *sendfile* in popular applications such as Apache and Samba.

The *sendfile* approach is nicely flow controlled by coupling file access to the network protocol: returning TCP acknowledgments drive reading more of the file into cache and possibly freeing already-sent parts of the file.

2.2 RDMA Complications

The benefit of moving data directly to the network, without intervening kernel buffers, comes with the cost of memory registration. In order to avoid these costs, application writers commonly use two strategies. The first is static registration: allocate a fixed buffer at initialization time and send and receive all data using that buffer. This, of course, introduces a data copy back into the critical path, and it has severe scalability problems in a server setting, as each connection or client will require its own private buffer.

A more realistic strategy is to perform memory registration inside the critical path, but to cache those registrations in the hope that they can be used again later. Caching is necessary to achieve reasonable performance, but implementing caching requires strategies on what to cache, what to release when the cache is full, and, in a library setting, how to notice when the application unmaps the cached region [15]. This usually results in some sort of least-recently-used eviction algorithm with various tunables and hooks to disable certain system calls.

Ironically, with the current state of affairs, serving file contents with RDMA will always require one memory copy: read the data into a buffer, register the buffer, then perform a zero-copy send to the network. Note that an application could go one step further and cache not only the registrations, but also the contents of the file being served. But since the operating system already implements a sophisticated file buffer cache that tracks usage and reads ahead based on access patterns, duplicating this in an application would be a bad design choice.

3 Design

We discuss three designs, increasing in their sophistication, to send file data using an RDMA network.

3.1 Naive approach

As discussed above, an application can read file data directly into a statically allocated buffer, at the expense of scalability, or it can read into any buffer and register just before sending it, with the considerable overhead of registration.

One can also use `mmap` on the file to have it appear directly in the virtual memory space of the process. This appears to avoid the copy, but, as the newly-mapped buffer is registered, the file contents will be faulted in so that the NIC can access the data. The semantics of memory registration in the current OpenFabrics implementation of RDMA requires that the file be opened with write permissions so that it can be registered as a shared, writable mapping. Otherwise, a full copy of the file cache pages will be performed into new memory to be mapped into the application. This interferes with most file serving implementations, where the daemon that serves the content runs as an unprivileged user (e.g. Apache) that only has permission to read files. Our experiments in Section 5 do not impose this additional copy burden when testing the `mmap` approach.

Since the bulk of the time in memory registration is due to following page table entries and converting virtual addresses to physical ones, the `mmap` approach is still quite expensive, even though it avoids a copy.

3.2 Pipelining

Another approach to gain an improvement in performance is to bring the file into user memory and to register that memory in a pipelined manner. This makes it possible to overlap sending of the data and copying of file contents to user space, at the same time keeping the network pipe full. Pipelining can also hide the cost of memory registration.

The number of pipelines, that is the number of simultaneous events occurring is a key factor in the performance. The other facet of performance is related to the depth of each pipe, meaning how much data is sent or copied at each stage in the pipeline. Tuning the depth is analogous to tuning of the send buffer in TCP.

While pipelining will improve performance for a single transfer, it does not remove the system effects of copying and registration, just overlaps them with network activity. The burden of copying on the memory bus, and registration on CPU cycles, will limit overall system throughput, especially in file servers that handle multiple clients simultaneously.

3.3 RDMA Sendfile

Taking things a step farther, it would be beneficial to remove the burden of copying and/or registration entirely. Borrowing the theme from TCP sendfile, where file data is moved directly from page cache to socket buffer, we designed a sendfile-like scheme that works with RDMA networks.

Combining the above notion of an RDMA sendfile operation with the pipelining scheme discussed above, we remove as much of the overhead as possible, and end up with a very efficient mechanism for transferring file data across RDMA networks.

In our design, we use the kernel to manage the file cache and register memory regions on behalf of the user process. It returns local access keys to the memory region that the user process submits in work requests to the NIC. It also unregisters the memory regions as requested by the user application. Because the kernel has the physical addresses of the pages that hold file contents, no page-table walking or translation is necessary. And because the user application need not see the file contents to ship it across the network, no copying or mapping into the user address space is ever performed.

As the file size can exceed physical memory or limits on pinned memory regions, we use an iterative, pipelined design to send the file in multiple chunks. The user application opens the file (but does not read it), initializes data structures, and loops over the file size. At each trip through the loop it invokes the kernel to register a section of the file and deregister previously completed sections, submits work requests to the NIC, polls for completed work requests. The user-space thread blocks while waiting for completion notifications, thus the whole process is naturally clocked by the network and consumes little CPU overhead.

One possible alternative design would be to have the kernel initialize the RDMA connections and manage the entire process itself. Then there would be no restriction on submitting work requests to a protection domain for a user-space application. However, this would involve moving a large amount of functionality into the kernel that does not belong there, including the entire state machine for connection management and error handling for failed connections. We did not pursue this approach due to the complexity and need for so much code duplication.

4 Implementation

For this work, we have implemented all three scenarios described in the preceding section. All approaches are implemented in a shared library that exports a sendfile-like interface to the user, allowing us to use identical test codes and to facilitate application deployment. Our RDMA sendfile software further uses a kernel module to handle memory registration of file cache data. The shared library handles all pipeline organization, as well as posting send requests and retiring completion queue events.

The software stack we use to communicate with the network infrastructure is OpenFabrics [10]. This allows our software to work unmodified either on iWARP or InfiniBand hardware. Minor changes are needed to the OpenFabrics stack for our current kernel module implementation: two symbols used to locate the kernel representation of a user protection domain are exported to our module. This small change will go away once our sendfile extension is integrated into OpenFabrics.

The user application specifies the limit on the number of pages to register at once which gives us a maximum message size based on the hardware page size. The user also specifies the number of outstanding operations, selected based on maximum completion queue depth and resource limits.

4.1 Security Concerns

As with TCP sendfile, the kernel has direct access to the physical pages that hold the file contents. Unlike in the TCP case, however, the kernel can not submit work requests to an existing connection. RDMA protocols, including InfiniBand and iWARP, were designed so that the OS would be bypassed in the send and receive path. To be more precise, InfiniBand verbs are defined very strictly to guarantee that memory of any particular process can not be accessed by another process. Queue pairs are created in a protection domain, and all memory accesses are constrained by that protection domain. The kernel also can not submit arbitrary work requests to queue pairs constructed by a user process. Given the well defined security model of RDMA interconnects, layering security protocols such as SSL on top of RDMA is feasible.

5 Results

For the experiments that follow, we used InfiniBand as the RDMA network. Each node is equipped with dual Opteron 250 processors, and 2 GB of RAM. The operating system is Linux, version 2.6.20. The Infiniband devices are 4X Mellanox single data rate, mem-free cards.

For software, we have coded a test driver that uses either our in-kernel RDMA sendfile implementation, or our pipelined user-space RDMA sendfile implementation. Both sendfile libraries implement the same pipeline scheme, and endure the same amount of memory registrations, polls to completion queue, and so forth. The main difference is that the user space library registers memory directly, while the kernel library uses the in-kernel version of the OpenFabrics memory registration verb to register physical pages with the NIC. The kernel module also must manage a list of pages for each pipe, freeing as necessary.

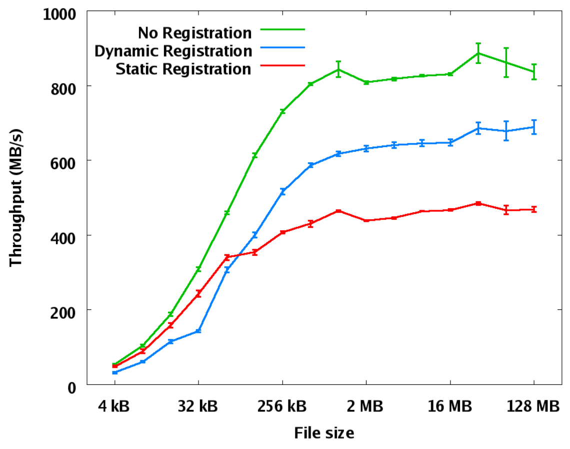

Figure 2. Naive approach.

We use InfiniBand in the following experiments mainly due to stability and hardware availability. A possibly better target would be a more wide area friendly interconnect such as iWARP. Despite the lack of iWARP hardware availability, we can implement on InfiniBand and the lessons learned will apply equally. A platform such as iWARP has the advantage that only servers need to be equipped with hardware, clients can make use of software [5, 6] to emulate the required network protocols. Our prior work with Apache is one such example [4], we outfit an Apache server with a hardware iWARP card and have multiple clients which emulate iWARP in software in order to realize RDMA benefits at the server.

Despite the obvious draw backs, InfiniBand is finding use in the wide area through the use of so called long-haul technology, and has even been demonstrated to be routable [9]. Given these relatively new advances it is not out of the question that we will see wide area usage of InfiniBand in the near future.

While our experiments tend to focus on performance rather than scalability, for our target applications such as web servers, an increase in performance will mean an increase in scalability. Higher throughput translates into handling clients faster, which means more clients can be handled in the same amount of time.

5.1 The Naive Approach

In Figure 2, we show the two ways to send file contents, static and dynamic registration, along with a curve to show the maximum throughput when performing no memory copies or registration. In static registration, the file is read into a pre-registered buffer. With dynamic registration, the file is mapped into memory, and the resulting buffer registered.

The top curve is the one generally shown in benchmarks. It does not include time to register or copy memory. For this particular curve, we have carried out these activities outside of the timing loop. With static registration, performance is good up through about a 64 kB message size, at which point the memory bus becomes the bottleneck. While the initial overhead of performing dynamic registration keeps its performance low at small message sizes, it is not bound by the memory bus and steadily increases with message size. However, the effects of registration are clearly visible even at large file sizes. This is due to the CPU-intensive nature of memory registration: each hardware page (4 kB usually) must be individually identified and manipulated. As the working set size of the relevant page tables exceeds the L2 cache size of the machine, latency for memory operations also becomes a significant bottleneck.

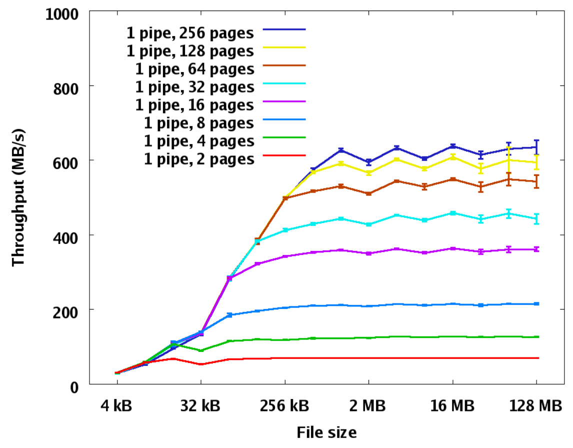

Figure 3. Page size effect on pipeline.

5.2 User-space Pipelining

Due to the shortcomings of the approaches discussed in the previous section, we now turn our attention to the benefits of pipelining the memory registrations. Figure 3 first shows the case where pipelining is disabled, just to illustrate that large message sizes are indeed advantageous, as they amortize both the sending overhead and registration overhead. This is the pure user space implementation, against which we will compare our kernel implementation in the next section. The upper limit of performance is the same as that in Figure 2 for dynamic registration.

Figure 4 shows the more interesting effect of increasing the number of concurrent operations. As few as two is sufficient to increase the throughput significantly. Higher values may be better, but at the cost of overheads in bookkeeping and registered memory footprint.

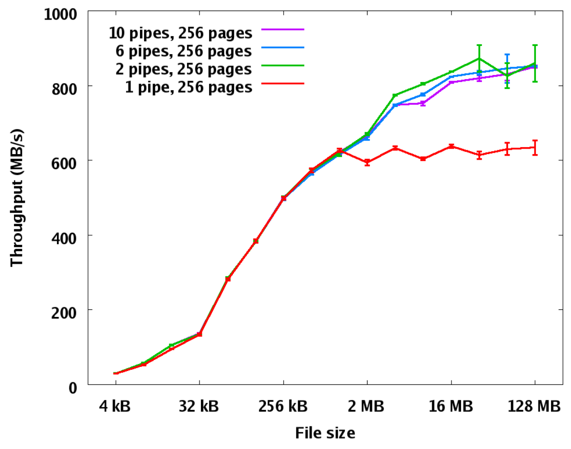

Figure 4. Pipelining depth.

5.3 RDMA Sendfile

In this section we show results from our new design that uses the kernel to identify and register file pages, while userspace continues to send messages and reap completions.

Just like with the user-space approach, we use pipelining here. The results shown in Figure 5 show the same sort of pattern as in Figure 3, in that larger messages achieve better throughput. Although in this case, the message size required to achieve a certain level of performance is smaller.

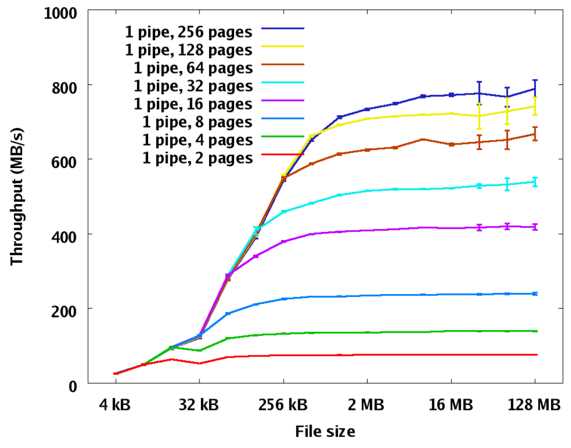

Figure 5. Page size effect on RDMA Sendfile.

Figure 6 shows the effect of increasing the number of concurrent operations, or “pipes”. Here, the marginal improvement by going from one to two pipes is very much smaller than in the user-space case, due to the lower overhead of the in-kernel RDMA sendfile approach.

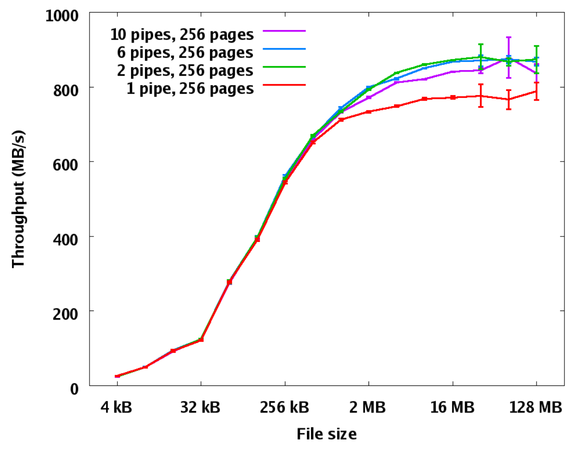

Figure 6. RDMA Sendfile pipeline depth.

5.4 Per-connection Memory Requirements

One of the ways to improve scalability for network applications is by limiting the amount of memory each connection is allowed to use. For instance with TCP, socket buffers often range from 32 KB to 64 KB per connection. In figures 7 and 8 we show the performance for sending a 16 MB file for pipeline depths of 1, 2, and 4. We do this for 4, 8, and 16 pages each. The total buffer size is calculated by multiplying 4 KB by the number of pages, and multiplying that by the number of pipes. For convenience the total buffer size in KB is calculated and shown above each bar. We see that even for small TCP like buffer size of 32 KB and 64 KB it is possible to achieve throughput in the range of 200 MB/s to just over 400 MB/s. By increasing the total allowed buffer size to 128 KB per connection, and utilizing 2 pipelines it is possible to achieve nearly 700 MB/s. If we continue to increase total buffer size to 256 KB per connection it is possible to achieve full throughput with the sendfile version and slightly more than 700 MB/s with the pipeline version. It is interesting to note that at small buffer sizes the importance of RDMA sendfile becomes quite apparent. This is especially true for 4 pipelines and 256 KB of buffer space.

5.5 Overheads

Our kernel-based sendfile design incurs some overheads, but achieves better performance due to avoiding virtual address translation. In the case where the application cannot open files for writing, it also saves a full memory copy of the data, but that effect is not shown anywhere in this paper.

The overheads in RDMA sendfile involve making multiple system calls to invoke the kernel, but these

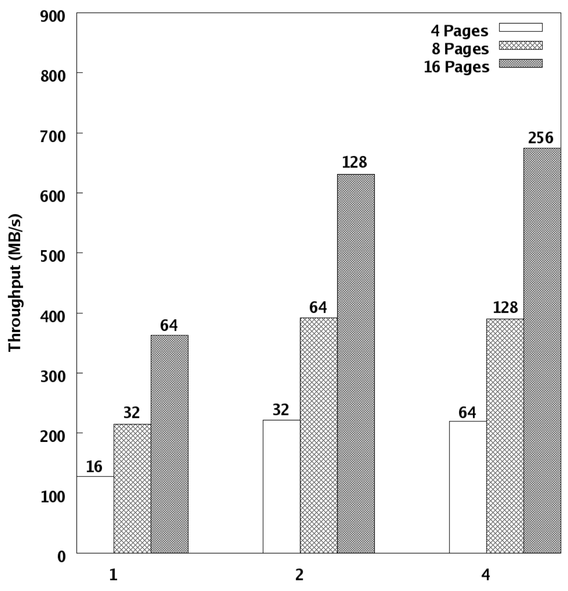

Figure 7. Pipeline performance at varying depths.

are very inexpensive. We use existing infrastructure for sendfile to look up the pages of the file, and this is done one page at a time, with a callback into our code. That overhead is also comparatively small, as measured by disabling various aspects of the code and comparing to the full version.

Figure 9 displays the major difference between user-invoked and in-kernel memory registration that we exploit in RDMA sendfile. The top curve shows the time to register a single memory region of a given size, and is similar to that found in previous analyses [7, 15]. The bottom curve shows the time for the kernel to register file cache pages. Memory registration consist of three operations: pinning down memory pages, translating virtual addresses to physical addresses, and transmitting the list of physical addresses to the adapter. At small message sizes, the time to communicate with the network adapter dominates [8]; however, for larger sizes the time to do the virtual address translation dominates. Both operations include the time to pin the pages in memory. The costs to deregister memory are identical in both cases. In the graph we see what looks like two outliers per curve, the first two data-points are either a hardware or firmware issue.

Memory registration, be it virtual or physical does not impact how long it takes the NIC to send a message. Keeping this fact in mind. The overhead from the registration, and the difference between the two overheads makes it easy to visualize the effect of memory registration on message latency.

Our implementation takes advantage of this differ-

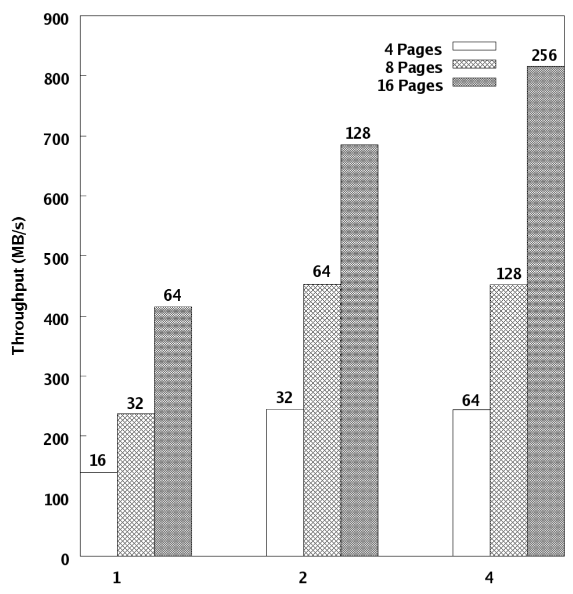

Figure 8. RDMA Sendfile pipeline perfor-

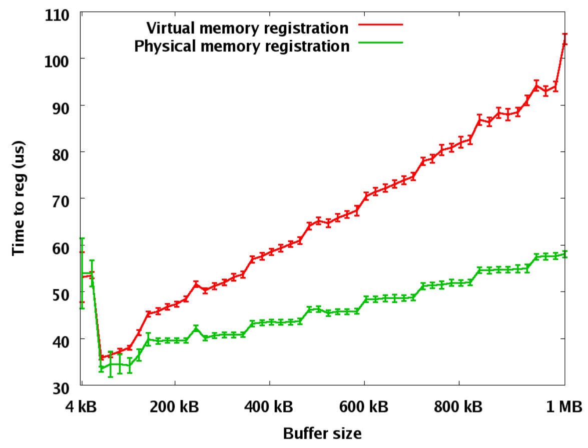

Figure 9. Cost of memory registration.

ence by avoiding the unnecessary mapping of file cache data into user virtual memory, then translating it right back into physical addresses.

5.6 Putting it all together

Now that we have explored the performance characteristics of each mechanism, we aggregate in Figure 10 a summary of each of the three mechanisms. All of these curves include the cost of memory registration in the transfer time, as is appropriate for file serving applications.

Both the in-kernel sendfile and user-space pipelined versions use 6 pipes and 256 pages per pipe (1 MB). The non-pipelined user-space code tops out at 700 MB/s, which although better than doing memory copies into a statically allocated buffer, is lower than what the hardware can provide and quite CPU intensive.

Interesting to note is that the performance of the pipelined and non-pipelined user-space code is the same up until around 1 MB, the size of a single message in the pipeline. Thus beyond 1 MB, the advantages of overlapping registration with sending is apparent.

The higher curve shows the in-kernel RDMA sendfile approach, which also is similar to the non-pipelined versions for small messages. Starting from 256 kB, the effect of avoiding virtual to physical translations can be seen, as discussed above.

These results hint that at even higher network speeds such as can be obtained with double- or quad-data-rate InfiniBand, or when using multiple iWARP NICs, the kernel-based RDMA sendfile approach will show even more advantage. Due to the reduction in CPU utilization and memory overheads, file servers will find it important to rely on the RDMA version of sendfile as much as they do on the TCP version today.

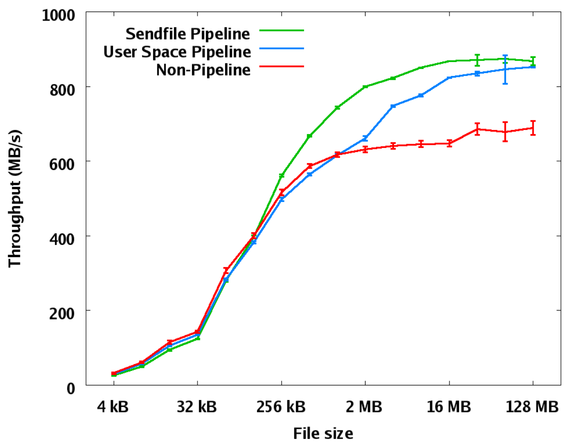

Figure 10. Comparing all file serving schemes.

6 Related Work

The costs associated with memory registration is examined in a number of works, including [15, 7, 3, 14]. The paper by Tipparaju et al. [14], explores three schemes for dealing with memory registration, two are similar to what we describe as dynamic, and static buffering, the third is an approach that gives users an interface to allocate already registered memory. Another approach to ease the burden of memory registration referred to as hugepages is described in [13]. Such a mechanism would only provide a marginal amount of improvement, as it does not address the issue of copying file data. Though the act of memory registration would be less costly, as there is less to register.

The importance of overlapping communication and computation is a thoroughly explored topic. One example which looks at issues related to pipelining is [2]. In this work it was shown that pipelining for large messages had little effect on performance, in contrast to this, we found pipelining was necessary for large messages. This is most likely due to the overheads of dealing with file data in our case.

Our approach to sending a file directly without copying the contents to user space is not a new idea. This was first proposed by Park et al [11]. They describe an extension to VIA [1] that lets a user specify a file to send, and the kernel provides the physical addresses to the NIC to do so, but they do not address how the kernel manages the cached pages or the use of pipelining. This was also done as an FPGA implementation on gigabit Ethernet.

References

[1] P. Buonadonna, A. Geweke, and D. Culler. An implementation and analysis of the Virtual Interface Architecture. In *Proceedings of SC 98*, Nov. 1998.

[2] A. Cohen. A performance analysis of 4X InfiniBand data transfer options. In *Proceedings of IPDPS 03*, 2003.

[3] D. Dalessandro and P. Wyckoff. A performance analysis of the ammasso RDMA enabled ethernet adapter and its iWARP API. In *Proceedings of Cluster 05, RAIT Workshop*, Boston, MA, Sept. 2005.

[4] D. Dalessandro and P. Wyckoff. Accelerating web protocols using RDMA. In *Proceedings of SC 06, Poster*, Tampa, FL, Nov. 2006.

[5] D. Dalessandro, P. Wyckoff, and A. Devulapalli. Design and implementation of the iWarp protocol in software. In *Proceedings of PDCS 05*, Phoenix, AZ, 2005.

[6] D. Dalessandro, P. Wyckoff, and A. Devulapalli. iWarp protocol kernel space software implementation. In *Proceedings of IPDPS 06, Communication Architectures for ClustersWorkshop*, Rhodes, Greece, 2006.

[7] D. Dalessandro, P. Wyckoff, and G. Montry. Initial performance evaluation of the NetEffect 10 gigabit iWARP adapter. In *Proceedings of Cluster 06, RAIT Workshop*, Barcelona, Spain, 2006.

[8] E. Mietke, R. Rex, R. Baumgartl, et al. Analysis of the memory registration process in the mellanox infiniband software stack. In *Proceedings of EuroPar 06*, 2006.

[9] Obsidian Research Corporation. http://www.obsidianresearch.com.

[10] OpenFabrics Alliance. http://www.openfabrics.org.

[11] S. Park, S. Chung, B. Choi, and S. Kim. Design and implementation of an improved zero-copy file transfer mechanism. In *Proceedings of PDCAT 04*, Dec. 2004.

[12] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/.

[13] R. Rex, F. Mietke, W. Rehm, et al. Improving communication performance on InfiniBand by using efficient data placement strategies. In *Proceedings of Cluster 06*, Barcelona, Spain, 2006.

[14] V. Tipparaju, G. Santhanaraman, et al. Host-assisted zero-copy remote memory access communication on InfiniBand. In *Proceedings of IPDPS 04*, 2004.

[15] P. Wyckoff and J. Wu. Memory registration caching correctness. In *Proceedings of CCGrid 05*, May 2005.