Memory Registration Caching Correctness

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

Jiesheng Wu
Ask Jeeves
170 Knowles Drive
Los Gatos, CA 95032
jwu@askjeeves.com

***Abstract*— Fast and powerful networks are becoming more popular on clusters to support applications including message passing, file systems, and databases. These networks require special treatment by the operating system to obtain high throughput and low latency. In particular, application memory must be pinned and registered in advance of use. However, popular communication libraries such as MPI have interfaces that do not require explicit registration calls from the user, thus the libraries must manage this aspect themselves.**

Registration caching is a necessary and effective tool to reuse memory registrations and avoid the overheads of pinning and unpinning pages around every send or receive. Current memory registration caching schemes do not take into account the fact thet the user has access to a variety of operating system calls that can alter memory layout and destroy earlier cached registrations. The work presented in this paper fixes that problem by providing a mechanism for the operating system to notify the communication library of changes in the memory layout of a process while preserving existing application semantics. This permits the safe and accurate use of memory registration caching.

I. User-controlled network interfaces

Computing systems have always had a variety of devices attached to them to interact with the external world, including local and wide area network adapters, disk drive controllers, and user input devices. These are almost always controlled by the operating system (OS), in the sense that the OS mediates all interactions of the device with the rest of the system. For example, when a user wants to read a block from a disk, the user invokes an OS interface that in turn issues a command to the device to put the block into a certain location of memory. It orchestrates all data motion and "knows" what parts of memory will be changed by outstanding device requests.

Recently, though, with increasing capabilities of external devices, this traditional OS-mediated interaction has become a bottleneck. Network interfaces in particular encounter severe limitations. While commodity hardware exists to transmit data at 10 Gb/s with latencies of under 5 μs, these rates cannot be achieved by a userspace process when accessing the device through the intermediary of the operating system.

Many network devices now exist that provide "OS bypass" mechanisms to permit users to manipulate the devices directly and invoke incoming and outgoing transfers. This is intended to reduce overheads and thus latency for small operations. Another capability is called "zero copy", indicating that there is no extra buffering between the network card and the user application buffers, unlike typical OS-mediated communication paths that include at least one extra buffer, thus one extra memory copy. Eliminating memory copies increases the effective throughput obtained by the application [1]. These intelligent network interfaces support many features to decrease the overhead incurred by the user application on each transfer, such as scatter/gather, channel and memory semantics, and atomic operations. Currently available network interface cards (NICs) that fall into this category are Myrinet [2], Quadrics QsNet [3], and InfiniBand [4], among others.

We describe implementation experiences with InfiniBand in this paper. The InfiniBand Architecture defines a system area network for interconnecting nodes that provide compute or storage resources. In its connected mode of operation, each InfiniBand NIC maintains a send queue and a receive queue for communication with the peer. As the user application posts work requests to the queue pair, the NIC performs the requests asynchronously and places status information, if requested, on a comple-

tion queue. Channel semantics require the receiver to prepost a receive descriptor before the sender posts the corresponding send. Memory semantics require only one side to be involved in the data transfer: Remote Direct Memory Access (RDMA) write operations move data into the memory of the peer directly and RDMA read operations fill local memory from peer memory, both without active participation of the remote CPU. Current hardware provides 1 GB/s of user data throughput with one-way small message transfer latencies of around 5 μs.

II. Memory registration

Users of zero-copy devices must coordinate with the OS to register memory regions needed for communication. The goal of memory registration is twofold: first, to allocate physical pages for process virtual memory and ensure they will not be swapped out; and second, to obtain the mapping of virtual addresses to physical addresses to provide to the NIC.

The InfiniBand kernel module and userspace libraries offer a set of basic calls to perform memory registration management. To register memory for use in later transfers, an application supplies a buffer address and length and receives in return an opaque handle that is used later to deregister the memory region. The registration call also returns two more opaque keys, one for local access and one for remote access, that can be delivered to other software components and other machines to permit them to access the memory region.

Figure 1(a) shows the effect of memory registration calls on basic InfiniBand performance. The upper curve shows the throughput obtained when all the memory is registered in advance of the test, and the lower curve includes the time to register and deregister each memory region during the test. The difference between these curves can be understood by examining Figure 1(b), which shows the time for the registration and deregistration calls. These delays, while small, are significant on the time scale of the transfer and cause low throughput values shown in Figure 1(a).

Because memory registration and deregistration operations are so expensive compared to the time to perform high-speed data transfers, many library and application writers are motivated to provide a cache infrastructure to avoid the need to deregister and later reregister the same memory regions. This works quite well in terms of performance, but the cache has many fundamental problems in light of the full Unix VM system. Caching is described in detail in Section IV.

III. Linux memory management

This section gives a brief overview of the behavior of a virtual memory system, and in particular, that of the Linux kernel, version 2.6.9. Many details are overlooked here for simplicity of presentation.

Virtual memory (VM) [5] is a mechanism to provide to each application the illusion of a full 32- or 64-bit address space but allow sharing of a significantly smaller amount of physical memory. It involves hardware support to divide physical memory into blocks (pages or segments) and to provide a protection scheme to restrict access only to particular processes. Hardware structures such as translation look-aside buffers and page-table walkers exist to speed up access to virtual to physical memory maps.

The use of VM in modern computing systems is so deeply ingrained that it would be highly unlikely that application programmers would give it up for the sake of speeding up network communications; hence, we consider in this paper ways to encourage the cooperation of user-controlled NICs and the VM system.

A. System calls

Traditional Unix semantics offer a variety of library and system calls by which a process can manipulate its virtual memory layout.

The system call `mmap` was first designed for mapping files into a process address space, but now is a general-purpose mapping facility that can be used for anonymous or shared memory, files, and hardware devices. This call has many parameters by which one can specify memory protection, suggested mapping location, file descriptor representing the target file or device, and extent of the range. To undo a mapping, the call `munmap` is used.

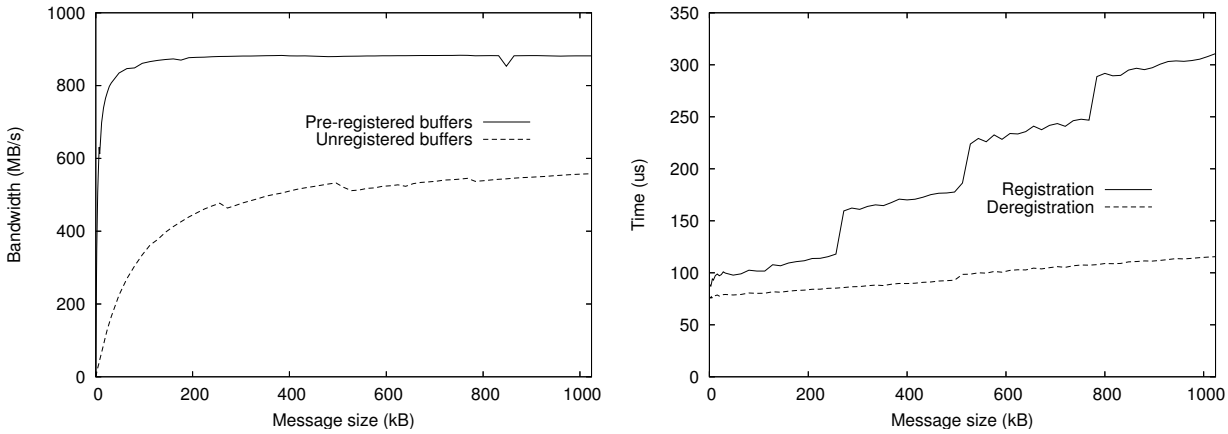

Fig. 1. (a) Impact of memory registration and deregistration on bandwidth. (b) Costs of memory registration and deregistration.

Specialized versions of these basic functions, called `brk` and `sbrk`, were used traditionally to change the data segment size. Now they essentially call `mmap` and `munmap` to manipulate memory around a certain point in the address space. Users generally use the library calls `malloc` and `free` that in turn invoke these system interfaces to change the VM.

Other less-often used calls can manipulate the VM space also. `Mremap` can enlarge, shrink, or move a region. `Mprotect` will change the protection bits of a region, regulating the ability of the process to read, write, or execute the contents. To inform the VM system that a region must not be swapped out to disk, `mlock` and `munlock` are provided as well as versions that affect all currently mapped memory. These usually require some sort of extra permissions not held by typical user processes. There are also `shmat` and `shmdt` to access system-wide shared memory, and various real-time POSIX [6] extensions to control typed memory objects.

Finally, process and thread creation and destruction events cause changes to the VM layout. In particular, `fork` and `vfork` create a new process by duplicating the VM of the current process. Most implementations use the copy-on-write technique to avoid copying all the process memory unless necessary. The Linux `clone` is a superset of process and thread creation with options to specify potential sharing of parts of the VM space by the newly created and original process. Implicitly, the operating system will further grow and shrink the stack of each process as needed, leading to another way for memory to be deallocated or remapped.

B. Memory pinning

As discussed in Section II, memory registration is needed both to translate virtual to physical addresses for the NIC and to ensure that that mapping remains constant across the lifetime of network operations involving the affected memory. The Linux VM provides an OS function that achieves both these objectives (`get_user_pages`), but it also provides to applications system calls that can change the virtual to physical address mappings, foiling any attempt at caching by a library.

An alternative option to pinning user memory is to provide preregistered regions allocated by the OS for the use of the application in communicating with the device. This is the approach used by other hardware-intensive systems, such as cameras and video frame buffers. It is not suitable for the types of applications that use high performance networking, though, because the APIs there have been developed to accommodate the use of arbitrary user memory in send and receive operations. Message passing codes that use, for example, MPI [7], a popular communication library, do not know until runtime which parts of the address space will be sent or received to other processes. This situation is also found in file systems that implement `read` and `write` without intermediate buffers, including DAFS [8] and PVFS2 [9].

Performance is another reason to require the use of an arbitrary part of the process address space: a file system read operation issued by the user

provides a destination buffer calculated at runtime. If the data from the network appears in a different pre-pinned buffer, a memory copy to move the data to its real destination must follow, but this extra copy considerably degrades overall performance.

IV. Memory registration caching

When the communication device requires memory registration but the API used by the application does not demand explicit registration calls, caching [10] is a common mechanism used to avoid the performance penalty illustrated in Figure 1(a). However, unexpected changes to the VM may occur due to application behavior not directly related to communications.

A simple way to avoid these unexpected changes by the application is to disable all calls that modify the virtual memory layout. Codes that are compiled statically and do not make VM-related system calls as described above lend themselves to an easy registration caching solution: at the first call to a communication library, look up the affected memory range in a private cache and register it if it is not found. Deregistration is not necessary unless the total number of registrations or amount of pinned memory exceed system-imposed limits. This class of code is becoming increasingly rare, but some older applications written in FORTRAN 77 [11], for example, would still work with this approach.

Since the bulk of VM activity is initiated by user calls to `malloc` and `free`, a common approach in the past has been to focus only on this subset of the overall problem. By replacing the standard C library calls at link- or run-time, a communication library can keep track of the state of memory affected by these calls and safely cache registrations of regions used for messaging. Calls to `free` by the application will cause the library to deregister the memory before releasing it to the operating system for reuse. This is the approach used by the Myrinet drivers, among others. There are two major problems with this approach: first, not all memory activity is initiated by calls to `malloc` and `free`. Of the list in Section III-A, some of the most common non-`malloc` activity found in applications are file mappings, shared memory, and process stack changes. The second major problem is that applications are constrained to use this one particular `malloc` implementation. Frequently for debugging or validation, programmers will use `malloc` replacements such as Electric Fence [12] or dmalloc [13] to track and display memory usage by their applications. This sort of debugging is made impossible by this approach.

For users of the GNU C library [14], its `malloc` implementation provides hooks by which a callback to a user-provided function occurs upon entry to any of the `malloc`-related routines. This permits some more flexibility by retaining the system-provided `malloc` but suffers the same problems described above plus a new one: the hook for `free` does not provide the size of the originally allocated object, requiring either another table to cache the size from `malloc` or probing internal structures of the `malloc` implementation to discover it. This is the approach used by mvapich [15], an implementation of MPI on InfiniBand. A different approach offered by the GNU C library is to use `mallopt` to restrict it to using only `sbrk`, not `mmap`, and never to return memory to the operating system. This avoids the `free` problem but adds serious restrictions to process growth in address space-limited architectures like the 32-bit Intel x86.

Another way to get "in front" of VM-modifying calls is to include symbols in the communication library that override the default weak symbols provided by the C library for `mmap`, `mremap`, and so on. This approach works, but not for calls initiated by the C library itself. So for instance, if `malloc` or any other routine that allocates memory on the user's behalf needs to use `mmap`, it calls into the C library internal symbol `__mmap` which can not be overridden. Furthermore, not all of the VM-modifying calls are exported via weak symbols, including `shmat` and `mlock`, and intercepting system calls in this way requires parsing all the parameters again since the behavior for `mmap` in particular varies considerably depending on its many inputs.

V. Linux VM interactions

The approach presented in this paper does not alter the `malloc` library or in any way restrict the activities of the user-space application—it may safely use the entire suite of VM calls provided

by the OS. Instead we take advantage of features in the Linux virtual memory subsystem to receive notification of "significant" VM changes in processes and manage cached registrations using that information.

Our module uses only currently exported public kernel interfaces. No modifications to a stock kernel tree of any sort are required. Restricting ourselves to use of just the exported interface makes module maintenance much easier as the kernel evolves as no patch is required for each particular release of the kernel. Possibilities for optimization that do need changes in the kernel core are discussed in Section VIII, however.

A. Linux data structures

There are a few major data structures that track the state of process memory allocations in Linux. Each process has a list of virtual memory area structures (VMAs). Each area describes a contiguous range in process virtual memory space and maintains flags for the area describing whether it is writable, executable, or locked among many others. The VMA has a pointer to per-area operations, including open and close, that are invoked when certain changes to this VMA have happened.

Each process also has a page directory that is the root of a two- or four-level hierarchy of other directories that are used to convert a virtual address into a physical address. These are almost always tied to the hardware mechanism supported by the architecture to perform address translation. The Linux VM system abstracts these tables and uses them both as a place to keep information on which pages are mapped to which processes and to alter the layout of virtual memory space for each process. The bottom-most entry in a page directory is a page table entry (PTE). Each PTE contains flags that can be set by the OS or by hardware to indicated if the page referenced by this PTE has been accessed or has been written, for instance.

B. Hardware driver

The main purpose of our driver is to track the effects the application has on the VM system in the kernel. It must also interact with existing drivers for the high-speed communication hardware. For the sake of discussion, we focus on current drivers for InfiniBand hardware, although similar mechanisms are used by most network implementations.

The InfiniBand driver library provides routines that a user application or communication library calls to register and unregister memory. These functions save a considerable amount of local state in userspace tables as well as in the kernel module, called "mosal," that actually does the locking. For our implementation we chose not to alter the mosal kernel-level driver or user library. Suggested improvements from integration are described in Section VIII.

The mosal driver uses two mechanisms provided by the Linux kernel to pin physical pages. The first is to increase the reference count in the page structure. The mechanisms that come into play during memory pressure are careful never to free such pages. They also set the `VM_LOCKED` flag in the VMAs of processes by calling essentially `mlock` on each region although this is not necessary for the current version of Linux.

VI. Dynamic registration module

Figure 2 shows the major components involved in memory registration. At the top is the user application that adopts a particular programming model and uses associated libraries to perform message passing. Frequently this is MPI, but other libraries that need communication may take advantage of InfiniBand directly too, including file systems such as PVFS2 [16]. To register memory, the libraries will call the vendor-provided mosal library that uses an associated kernel module to pin pages in the kernel's core virtual memory subsystem. A second path from the library to the VM core, on the left of Figure 2, shows our new component, called "dreg," that uses features provided by the VM to manage existing registrations accurately in the face of changes to the address space of the application.

A. Library routines

Messaging libraries that use dreg for caching use calls that are very similar to the underlying InfiniBand ones:

```
int dreg_register(void *addr, size_t len,
  int acl, u32 *lkey, u32 *rkey, u32 *mrh);
int dreg_deregister(uint32_t mrh);
```

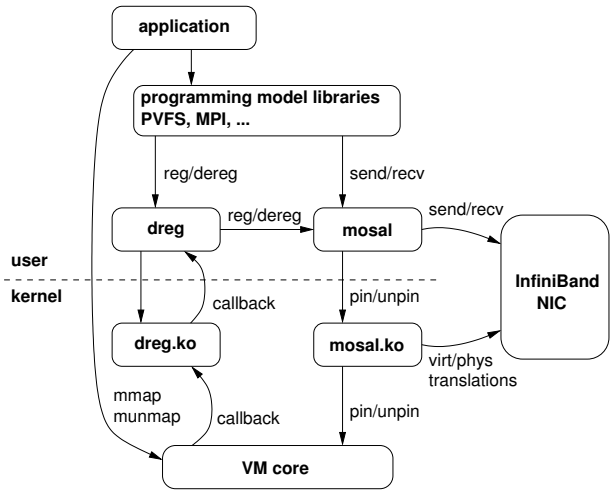

Fig. 2. Interactions among communication components.

The registration call takes an address, the length of the region, and access-control parameters specific to use by InfiniBand drivers. It returns opaque 32-bit cookies for the local key, remote key and memory registration handle. This handle is later provided to the deregistration call to identify the region, in part because overlapping segments are allowed by the underlying device.

The dreg userspace library calls the original code in mosal to pin the pages and inform the network card of the new translations and keys. Then it calls the dreg kernel module to register the new region with the VM subsystem. The interface between the user and kernel components consists of read and write calls on a character device allocated at module insertion time.

The other half of the functionality of dreg is to watch for VM changes in the application and report them back to the registration cache. There is one synchronous call:

```
int dreg_check(u32 *mrh);
```

that polls for changes and, if any are found, returns a registration handle that must be invalidated. This might be called at entry to the messaging library such as before sends and receives are posted. Another way to use the dreg library is to allocate a separate thread to wait for VM changes. For instance, a cache component that manages all the registrations for multiple libraries and NICs can provide a function that will take the necessary steps asynchronously as driven by the kernel module that monitors the VM subsystem. The messaging library must be sure to provide locking around the use and modification of registrations in this scenario.

Two uncomplicated routines, `dreg_open` and `dreg_close`, are used to initialize and finalize the library. Initialization requires opening the character device provided by the kernel module, and finalization closes it, triggering release activity of any regions still managed by the kernel.

B. Kernel module

The kernel module has functions to listen to the user library component as discussed above and also has functions that are registered as callbacks with the core virtual memory subsystem. Multiple applications, either independent or cooperating, can use the module at once; separate internal structures are used for each distinct memory management space. As discussed in Section V, each range in the virtual memory space of a process has an associated VM area structure (VMA). The dreg kernel module keeps track of registrations that the user library component tells it about and it keeps track of the VMAs that underlie these registrations.

When signaled through the character device from userspace, the module looks up the supplied address for the corresponding VMA. The dreg module takes note of the existing VMA callbacks and replaces them with its own callbacks. A new structure to hold information about the region is created and linked onto an internal list of registrations. Deregistration is a fairly simple undoing of these events.

There are two relevant routines in the array of callbacks in the VMA structure: `open` and `close`. The kernel calls `open` when it creates a new VMA and it calls `close` when it prepares to discard an existing VMA, passing the affected VMA as the only parameter. There is no callback for other changes in VMAs, just their creation and destruction. Fortunately this is sufficient to discover all the changes that might happen to existing registered memory regions in a VM. Calls to `open` and `close` can be initiated by many system calls, such as `mremap` and `munmap` (from `free`), but all these end up either destroying a VMA completely and calling `close`, or splitting an existing VMA by first calling `open` on a new one and then calling `close` on the old one.

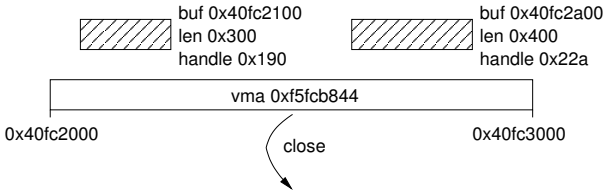

Fig. 3. VM structures when freeing an entire memory range.

C. Examples

A simple example of two registrations in a single memory region is shown in Figure 3. There are two registered buffers associated with a single underlying physical page. When the user code frees the buffer that contains that page, the C library will use the `munmap` system call to remove that area from the process address space. The kernel will undo all internal state required to perform this unmapping, then as the final step, invoke the `close` callback on the VMA to invoke the dreg module. It in turn notices that there are two registered regions associated with the VMA and adds two deregistration entries in a queue for notification to userspace. If a thread is blocking in a read on the character device, the module will wake it up so that it can process the new information immediately, otherwise the contents of the queue will be read later when polled.

The more complex example in Figure 4 shows what happens when a subset of a pinned memory range is freed. The system call that initiates this sequence of events is `munmap` of the middle of the three pages shown in the figure. In response, the VM system provides three separate notifications to the dreg module. First it splits the existing VMA by shrinking it (no callback) and opening a new one below the start point of the unmapping. Then it again shrinks the initial VMA and opens a new one above the end point of the unmapping. Finally it closes the isolated range as requested by the `munmap` call. The dreg module queues the appropriate deregistration notifications to userspace as the opens and closes change the existing mappings. Two of the mappings are revoked because they overlapped the closed region, while the third survives.

VII. Performance

The main benefit of this work is the introduction of consistency into the dynamic registration process and the ability to use the entire set of virtual memory manipulation system calls safely. In this section we show the costs and benefits of adding this consistency.

First, the overhead of making the extra calls to the dynamic registration module are very small. A plot of the modified registration and deregistration times on top of the graphs in Figure 1(b) produces indistinguishable curves and is not shown here. At no point is the difference more than 2% away from the base case.

To detect when the application has unmapped pages outside of the communication library, it is necessary to call into the dreg module periodically. This can be done either by using a non-blocking call, likely just before each use of the cache, or by devoting a thread to block for events and update the cache accordingly. The difference between doing explicit deregistrations versus calling the blocking `dreg_check` call to process memory change events is less than 1% for small buffers and disappears beneath the measurement error for buffers larger than about 100 kB. The other way to handle process memory changes is to devote a separate thread. This case is found to be *faster* by up to about 15%, provided there is sufficient processor idle time, such as when waiting for communications to complete. This is because processes often block while waiting for network or disk operations, permitting other threads to run in the background and effectively hiding the deregistration operation. When the test is configured to use all available processor cycles (on a uniprocessor machine with a constant busy loop), the overhead of thread switching can be seen to add an extra 2% to the total elapsed time of the benchmark. In a true application that performs other calculations, these differences are found not to be measurable.

Finally, we describe the real-world application that in part motivated this work. A material dynamics code was observed to crash after a large number of iterations on an InfiniBand cluster using MPI when it failed to allocate memory. It uses a C++ framework for message passing that dynamically allocates and frees memory for communication at each iteration. Their framework implements its own allocator that acquires memory directly from the

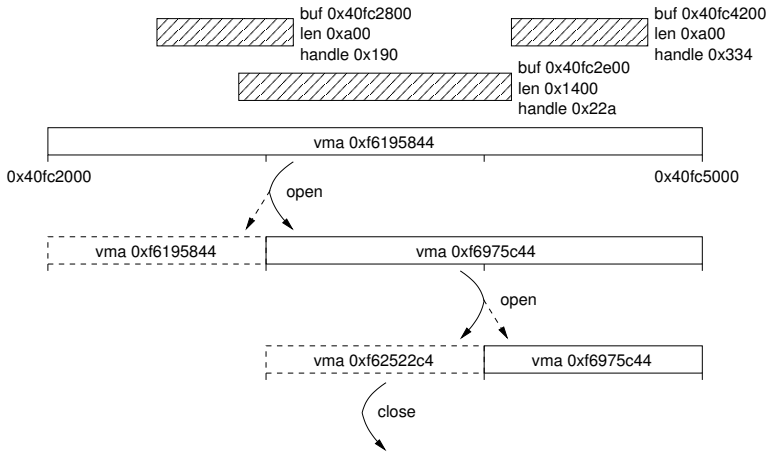

Fig. 4. Existing VM structures to illustrate freeing a subset of a pinned memory range.

system via `mmap`, thus avoiding the `malloc` hooks we had configured in the MPI library startup code. When these memory regions are passed into the MPI library, they are registered and cached in the hope that they will be used again; however, the memory is then immediately freed and in this case returned to the operating system. Since the InfiniBand driver had locked the pages in these dynamic buffers but never learned that the application later freed the pages, they were effectively leaked: the OS was unable to reuse them while they were held by the network driver. Using the dreg module and library for allocations resolved the problem.

VIII. Conclusions and future work

We have developed a mechanism by which the operating system can notify communication libraries of changes in the memory layout of processes without requiring changes to existing application semantics. This mechanism is necessary to achieve full performance of high-speed communication networks while retaining all the safety and functionality of existing Unix virtual memory systems.

As discussed above, there is some code duplication and inefficiency with the current implementation of the kernel module. We plan to offer a patch to repair that through small changes to the core Linux virtual memory system and more extensive changes to the NIC libraries.

Application libraries that use this consistent dynamic registration system still need to implement caching, that is, remember the keys and handles returned by calls to `dreg_register` and monitor the cache to ensure it does not grow too large. The optimization parameters for the caching problem may be different for each library and each application, though. We plan to develop general strategies to help with this general caching need.

References

[1] P. Druschel, "Operating system support for high-speed communication," *Comm. ACM*, Sept. 1996.

[2] Myricom, Inc., "Myrinet," http://www.myri.com.

[3] Quadrics, Ltd., "QsNet," http://www.quadrics.com.

[4] *InfiniBand architecture specification*, InfiniBand Trade Association Std. 1.2, Oct. 2004.

[5] D. Patterson and J. Hennessy, *Computer architecture: a quantitative approach*, 3rd ed. M. Kauffman, 2002.

[6] *Portable Operating System Interface (POSIX)*, IEEE Std. 1003.1, 2004.

[7] *MPI: A Message-Passing Interface Standard*, MPI Forum, Mar. 1994.

[8] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, *et al.*, "The direct access file system," in *Proceedings of FAST'03*, San Francisco, CA, Apr. 2003.

[9] N. Miller, R. Latham, R. Ross, and P. Carns, "PVFS2 for clusters," *ClusterWorld*, Apr. 2004.

[10] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa, "Pin-down cache: A virtual memory management technique for zero-copy communication," in *Proceedings of IPDPS 12*, Orlando, FL, Mar. 1998.

[11] *American National Standard FORTRAN*, ANSI Std. X3.9, 1978.

[12] B. Perens, "Electric fence," http://perens.com, 1995.

[13] G. Watson, "Debug malloc library," http://dmalloc.com, 2003.

[14] Free Software Foundation, "GNU C library," http://www.gnu.org/software/libc, 2001.

[15] J. Liu, J. Wu, S. Kini, P. Wyckoff, *et al.*, "High performance RDMA-based MPI implementation over InfiniBand," in *Proceedings of ICS'03*, San Francisco, CA, June 2003.

[16] J. Wu, P. Wyckoff, and D. Panda, "PVFS over InfiniBand: design and performance evaluation," Proceedings of ICPP '03, Oct. 2003.