Firehose: An Algorithm for Distributed Page Registration on Clusters of SMPs

Christian Bell Rajesh Nishtala
May 2004: CS262B Final Project
Computer Science Division
University of California, Berkeley

{csbell,rajeshn}@cs.berkeley.edu

Keywords: Memory registration, Remote DMA, Global-Address Space Languages, GASNet, PThreads, Firehose, High-performance networks, Myrinet.

Abstract

This paper proposes to improve a memory registration strategy for Remote DMA operations over pinning-based networks in the context of Clusters of Multiprocessors (CLUMPS). Although existing approaches focus primarily on bandwidth as a metric for evaluating the cost of DMA page registration (or pinning), there are many levels of host synchronization that hide the true cost of registration and prevent efficient use of one-sided RDMA communications and seriously impacts the cost for small messages. Furthermore, existing approaches do not tailor their implementations for Multiprocessors that share a single network endpoint, and as such ignore to exploit the spatial and temporal locality in the remote memory pages referenced by running many SPMD parallel programs over a single network endpoint. The proposed solution extends the idea of the Firehose algorithm and proposes a new Firehose API and library for both uniprocessor and multiprocessor variants. The SMP-aware version of Firehose pursues the same design goals of minimizing host-level synchronization – or more specifically, allowing operations to complete one-sided in the common case and reverting to rendezvous-like synchronization between endpoints when operations in the uncommon case. We motivate the need for special attention for CLUMPS and supplement a complete Firehose-SMP implementation with an evaluation of its design and some of the initial performance results are promising.

1 Introduction

User-level Remote DMA communication is the preferred approach for delivering high bandwidth and low latency data transfers in most high performance networking hardware today. Among some of its benefits is the ability to leverage zero-copy operations, which implies that the Operating System software stack and networking hardware agree on the protection mechanisms to allow users to read and write directly on the network. Vendors such as Myricom and Quadrics [2, 5] have leaned on this technology for years and offer relatively low-cost high performance network interconnects for loosely coupled Networks of Workstations. Although they differ in the level of integration of the NIC with the host, vendors are always actively working on moving an increasing amount of functionality onto the Network Interface. In this regard, support for Remote DMA operation is seen as the fundamental building block for most network operation enhancements.

Each of these technologies ships with some version of the ubiquitous Memory Passing Interface (MPI), which many regard as the ultimate standard in High Performance Computing. Recently, MPI's dominance has been countered by a new breed of Partitioned Global Address Space (PGAS) languages, who's development effort was motivated by the need to provide a richer set of language-level parallel programming constructs. For example, Unified Parallel C (UPC) promotes the high level of programmability of shared-memory systems while still underscoring the need for performance by giving the user control over data layout. A key difference in terms of the memory requirements of MPI and PGAS languages is the need to expose the entire virtual memory address space in the latter, much like shared-memory platforms. Existing technologies in the area of high performance interconnects reveal various implementation efforts and in many cases, limitations as to exposing the entire address space for RDMA have important performance impacts.

The recently proposed firehose algorithm [1] attempts to address some of these limitations and is designed in the spirit of GAS languages. Namely, Firehose allows zero-copy one-sided operation in the common case, exposes the entire virtual memory space and is sensitive to small message sizes. The first version of the algorithm was implemented as an integral part of GASNet's GM conduit, in order to target the page-based pinning interface offered by Myrinet. The latest technologies (such as Infiniband) add to the amount of pinning-based networks and also increase the possible firehose clients. After giving an overview of firehose in section 3 This paper first proposes, in section 3.3, an interface and library that can be targeted by various flavors of pinning-based networks, namely page-based and region-based. The remainder of the section raises some of the issues present in maintaining a consistent firehose state table in the presence of split-phase multi-threaded accesses and updates and assess the need for an SMP awareness in firehose. The paper follows with a description of the design and implementation of firehose SMP in section 4 and presents some initial performance and correctness results in 5.

2 Background

Support for Remote Direct Memory Access (RDMA), available in many high performance Network Interfaces, allows the NIC to write and read directly to memory without interrupting the host processor. Memory bandwidth aside, the benefits of this approach for the host processor are clear – the cache is not polluted with the contents of bulk (possibly useless) data and the host processor can concentrate exclusively on computation and not servicing of the network. In terms of support for RDMA, there are two facets to the problem. The first involves supporting cache invalidations between a DMA interface and the host processor, which processor architectures already support for other legacy DMA devices. The second requires the NIC to handle the virtual-to-physical mapping of memory pages as well as remain synchronized with the state of the host's page tables so that pages destined and sourced for DMA are guaranteed to be present in physical memory. This point separates two classes of network interfaces in their support for page registration:

- **Hardware-assisted**. This approach is not unlike a regular paging-based virtual memory system except that the NIC combines a hardware TLB and tweaks in the kernel's virtual memory subsystem which allows the NIC to pin pages, initiate page faults when necessary and track changes in the application's page table (see Quadrics [5]). Since this approach poses no restrictions on what memory is made DMA-able, potentially all of the user's virtual memory can be made available to remote DMA operations. The downside, however, is the higher price tag for these NICs and the custom modifications required for every OS the NIC supports.

- **Pinning-based**. This approach requires the programmer to explicitly set up the regions of memory to be enabled for DMA operations. This translates into marking the relevant memory pages as non-pageable (referred to onward as pinned) in main memory. Pinning user-level virtual memory pages instructs the OS that the underlying physical pages cannot be paged out until the application terminates or explicitly unpins them. Due to this restriction, the upper bound on the amount of memory that can be pinned at one time (and therefore made available for remote access) is limited by the size of physical memory (in practice, the limit is actually somewhat less than physical memory size, depending on the OS and NIC hardware). This restriction is especially problematic in 64-bit applications with large memory requirements where the total virtual memory space in use may far exceed the physical memory size (although the actual working set may be rather small).

Aside from hardware-assisted approaches which can implicitly handle memory registration, existing software approaches to dynamically and explicitly pinning memory are insufficient. In order to support the more interesting features leveraged by GAS languages, allowing one-sided remote memory operation and exposing the entire virtual memory address space are two major design goals in developing a pinning-based RDMA strategy. Also, since message sizes in languages such as UPC tend to be much smaller than in MPI, providing low latency by way of remote RDMA is also desirable. Existing approaches to dealing with memory registration are summarized in table 1. Aside from Firehose, none of the software approaches provide zero-copy one-sided operation over the Full VM, and none of them have tuned their implementations for CLUMPS.

Strategy	*Zerocopy*	*Onesided*	*Full VM*	*SMP aware*	*Notes*
Hardware-assisted	√	√	√	N/A	Hardware complexity and price, Kernel modifications
Pin Everything	√	√			Limited memory, May require a custom memory allocator
Bounce Buffers			√		Two-sided, Local copy costs (CPU consumption), Messaging overhead (metadata and handshaking protocol)
Rendezvous	√		√		Two-sided, Registration paid on every operation
Firehose	√	√	√		Zero-copy, One-sided (common case), Full memory space accessible, Only handshaking is synchronous, Registration costs amortized. Messaging overhead (metadata and handshaking protocol) on firehose miss (uncommon case)
Firehose SMP	√	√	√	√	The same as non-SMP firehose, but accepts and satisfies pinning requirements for multi-threaded clients with split-phase accesses

Table 1: Summary of available DMA registration strategies

Previous work with pinning-based DMA registration has involved optimizing performance of remote memory operations using strategies adapted to the underlying network hardware. These are formulated according to the method for posting communication buffers, how regions of memory are enabled for DMA, flow control and low-level network layer overhead. It has been shown that there is merit in considering various approaches to optimizing remote memory operations on pinning-based networks. In particular, [4] proposes two ways to deal with registration as a required component for remote memory operations: either by pinning/unpinning memory locations as part of each data transfer or by streaming data through preallocated, registered memory buffers. Depending on the underlying network parameters, one or the other is shown to provide better bandwidth. Some firehose performance numbers are shown at the end of section 3, after a description of the algorithm.

Clusters of SMPs have the particularity of sharing a single network endpoint, the node, and have the potential to benefit from a hybrid of sharing local low-latency memory with a high bandwidth network interconnect. Possible execution models on a each SMP nodes be can either pthread-based or process-based. The former benefits from a boost in performance for read-only portions of code and data shared between pthreads and sharing large amounts of memory between threads is implied as they live in the same VM process. In the process-based model, each processor must have its own virtualized context within the network endpoint and large amounts of SysV shared memory is not well supported across platforms. However, processes allow users to link with non thread-safe scientific computing libraries without trouble, which is not possible using threads. The debate on threads versus processes is the subject of further research as there are many other pros and cons for choosing one or the other in a High Performance Computing environment. This paper believes that pthreads is a valid execution model on a CLUMPs and examines the requirements of providing a general pinning-based strategy. Since target applications are typically SPMD parallel applications, there is reason to believe that there is an important amount of reuse in the particular remote pages that are referenced between parallel computation threads connected to a single network endpoint.

The previous implementation of the Firehose algorithm paid particular attention to low latency minimized the time spent querying the firehose table, and provided correctness and performance for split-phase local and remote accesses to the table. However, in the presence of multi-threaded split-phase accesses, many of the assumptions about the consistency of the table must be reexamined. The problem is much like providing a fully-reentrant Virtual Memory Manager (VMM) where hardware operations on the processor page table **cannot** be carried out atomically. Although this comparison bears on no practical system, the point is the VMM would have to deal with processes (or clients) wishing to reference pages which are in inconsistent states – pages can be committed in which case they are in memory or paged out to disk, or they can be in a visible yet inconsistent state during which they are neither in memory or on disk. For this latter case, the VMM would have to ensure correctness in the way these types of pages are handled by its clients.

3 Firehose Algorithm

3.1 Algorithm Description

The Firehose algorithm starts by determining the largest amount of application memory that can be pinned. This constitutes the upper bound on the total number of physical pages that can be simultaneously pinned and is generally a function of the size of physical memory. In order to prevent the application from swapping on its memory references to non-shared memory and respect the memory requirements of other running processes and the kernel, this value is limited to some reasonable (tunable) fraction of physical memory.

If this amount corresponds to a total of M bytes using P byte pages, then a total of M/P pages can be pinned at any time during execution. Since a node must support incoming remote memory operations from any other node in a parallel job, the available space can be evenly divided and $f = \lfloor \frac{M}{P*(nodes-1)} \rfloor$ physical pages can be guaranteed to each remote node. A firehose is a conceptual handle to a remote page and each node owns f of these firehoses to every other node. A node has total control over the fixed number of firehoses it owns, and is free to use any or all of them to establish mappings to remote pages (pinning those remote pages) in order to satisfy pending remote memory operations.

Once a node has properly situated one of its firehoses, mapping it to a region in remote virtual memory (via a round-trip synchronization message), the remote node guarantees that virtual page will remain pinned for the duration of the mapping. The requesting node can now freely "pour" data through the hose to or from that region of remote shared memory, in the form of one-sided remote DMA puts and gets. A firehose can be efficiently reused for multiple subsequent operations to the given region, exploiting the temporal and spatial locality of application memory references to amortize setup costs over many operations. As such, the Firehose algorithm is a distributed strategy for managing pinned memory. Figure 1 portrays a typical runtime snapshot of how two nodes use their firehoses to map selected remote pages on one node.

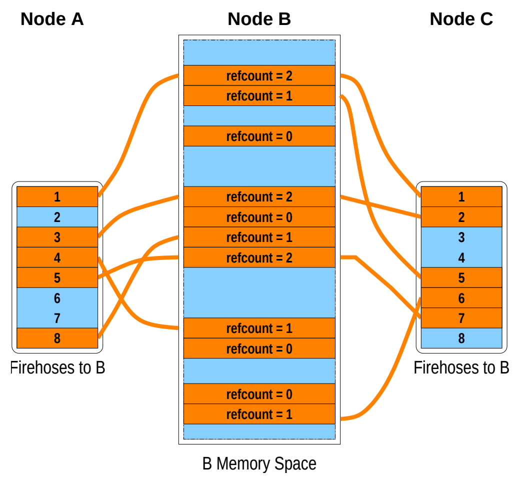

Figure 1: Runtime snapshot of two nodes (A and C) mapping their firehoses to another node (B)

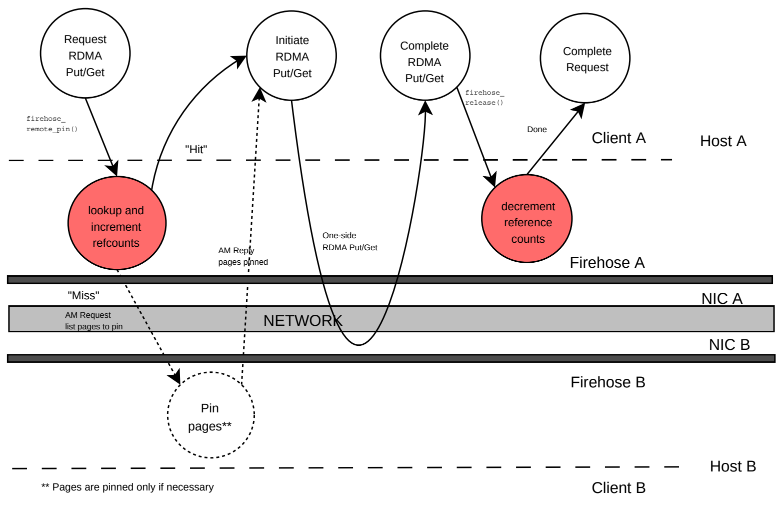

Figure 2: Firehose remote pinning flow

Implementation of the Firehose algorithm requires a thin control layer (such as Active Messages [6]) for the handshaking that takes place when a host wishes to move a firehose [1].

[1]Active Messages are an integral part of the GASNet API [3]

A remote memory request to pin memory is shown in figure 2. The steps followed are typical for a client that pins memory in order to issue a one-sided put (or get) operation.

1. The remote request on host A consults a table of firehoses for existing mappings to the remote node. If the destination memory is fully mapped by firehoses (i.e. a firehose "hit"), the put can be completed entirely with one-sided remote DMA (notice how the DMA only reaches the NIC on node B and node CPU B); if not, the second step follows. In either case, reference counts are incremented to declare on intention on the remote pages.
2. For a miss, a firehose move active message request is sent, which communicates a reassignment of firehoses. In general this involves moving of firehoses (by updating state metadata) - releasing old mappings which are not being used in favor of new ones.
3. Upon receiving a firehose move request, the virtual pages being released (if any) are unpinned and the new set of pages is pinned. A reply confirming the pinned destination memory is sent.
4. The one-sided DMA put or get operation may be sent (again, the operation is truly one sided).
5. Upon completion, a client typically releases its intentions on remote pages, which decrements the associated firehose reference counts and helps Firehose track which firehoses are actively in use on a remote node.

If the contents of figure 2 is considered without the darker circles in the firehose A layer of the diagram, firehose is essentially a rendezvous algorithm. This has been stated before and also constitutes the "miss" path taken by a request that cannot be fully satisfied by existing remote node firehose mappings. The above series of events represents an unpolished version of the algorithm. There are many potential optimizations and implementation details in dealing with firehoses, both on the requesting and receiving node:

- Each node associates a reference count with every locally-pinned page to track usage. While the reference count is greater than zero, this page is not a candidate for unpinning because it is currently in-use by one or more incoming firehoses or locally-initiated operations. While the page is pinned, subsequent remote requests to attach a firehose to this page or locally-initiated operations needing this page merely increment the reference count and incur no registration overhead (this situation is actually quite likely, especially in collective operations such as broadcast or reduce). The local node does not need to explicitly track which remote nodes have mapped a firehose to a given local page, because all incoming firehoses are entirely controlled by the remote node, and therefore we rely on those nodes to cache their active mappings;
- Firehose supports lazy deregistration using the page-based reference count. By allowing a configurable number of 0-refcount pages to remain pinned in memory, much of the burden of unpinning and re-pinning is sidestepped. Although this may lead to increased physical memory consumption, it has the potential to greatly reduce pinning overhead as a page lazily kept pinned may be the target for subsequent firehose moves or local operations. Networks with a high registration cost should be configured permissively with the lazy unpinning parameter;
- A victim FIFO queue tracks pinned pages with zero reference counts. Victim pages are evicted when firehose move requests arrive for pages not currently pinned or when we've reached the configurable limit of physical pages in use for pinning;
- A reference count is also tied to each firehose. The count is incremented when an local operation is initiated through the firehose and decremented once the operation completes. This count prevents race conditions between concurrent or overlapping operations that need to establish new firehose mappings.

Table 2 summarizes the data structures used to implement the Firehose algorithm with all these optimizations.

In order to be adaptive to various networks and memory configurations, the Firehose algorithm has the following tunable parameters:

Data Structure	*Description*
Local Page Table	• Table keyed on page virtual address, with one entry for each currently pinned page • Each entry contains a page reference count and pointers for the page Victim FIFO (doubly-linked list) • Reference count reflects locally-initiated RDMA operations in-progress for the pages of this page (e.g. source of a locally-initiated put) and the number of remote firehoses mapped to the page • Table can be implemented as a simple array on 32-bit platforms
Firehose Table	• Hash table keyed on tuple of remote node and page virtual address with one entry per attached firehose • Each entry contains a reference count, a reverse mapping to the tuple and a pointer for the Firehose Victim FIFO (singly-linked list) • Reference count reflects the number of locally-initiated operations in-progress which touch this remote page and are therefore using the firehose

Table 2: Primary data structures in Firehose implementation

1. **Maximum amount of physical memory used for remote firehoses (M)**
   This parameter limits the amount of memory the algorithm guarantees to remote nodes as pinnable at application startup (M). This amount of memory is consumed if and only if every remote node uses up all of its firehoses to a given node. It is likely that the upper bound for this value is limited by the network or other operating system level requirements.

2. **Maximum size of page victim FIFO queue (MAXVICTIM)**
   This parameter has been explained previously for its benefits in minimizing the registration overhead for networks where either the pin or unpin operations are expensive. When a local page's reference count reaches 0, it is added to the head of the victim FIFO queue - when the queue length exceeds this configurable parameter, pages are removed from the tail of the queue and unpinned.

The total physical memory usage of Firehose (i.e. pinned memory managed by the algorithm) never exceeds the upper bound of M+MAXVICTIM (so the sum should be restricted to be some reasonable fraction of the physical memory size). M is the maximum amount of memory that can be pinned at any time as a result of remote firehose requests, and the victim FIFO can additionally keep up to MAXVICTIM pages which aren't committed to a remote firehose (to reduce the cost of repinning them later, benefiting from temporal locality). Local pin operations (i.e. source of a put or destination of a get for pages not already pinned) can be satisfied by stealing some pages off the victim FIFO (or simply pinning new pages and later returning them to the victim FIFO if it's below the length limit). In any case, the local node is guaranteed at least MAXVICTIM space for local pin operations to unpinned pages, and the algorithm will still never exceed the M+MAXVICTIM hard limit (although it's expected to rarely reach this limit in practice).

3.2 Comparison with other DMA pinning strategies

The firehose approach for pinning memory was previously compared to other DMA registration strategies in table 1. The performance that firehose can attain in terms of bandwidth is shown in figure 3, where 64Kb randomly distributed "put" operations are sent from two nodes as part of an increasing target memory space [2]. Until the active working set reaches M, firehose provides a 100% hit rate and "put" operations can complete without any handshaking and at full network potential. As means for comparison, two versions of the rendezvous protocol are

[2]Performance results presented in this paper were obtained on an 80-node Myrinet 2000 Cluster, with dual Pentium III 866 Mhz/1GB RAM nodes running GM 2.0.11 and the latest CVS GASNet snapshot

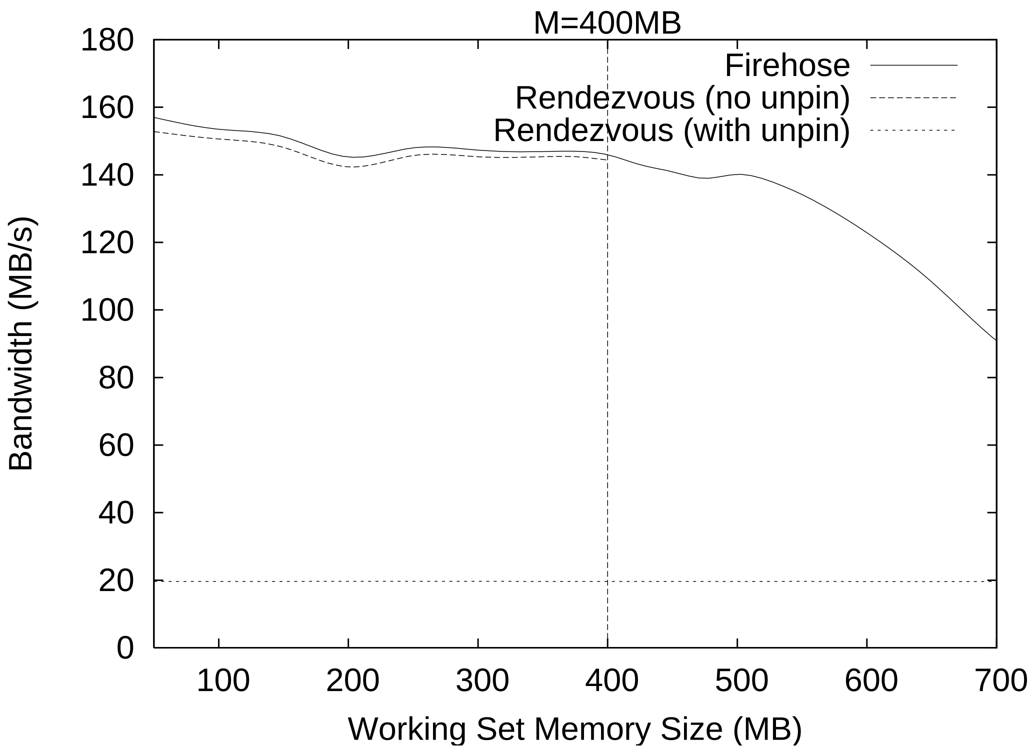

Figure 3: 64KB put bandwidth over increasing working set memory size (M = 400MB and MAXVICTIM = 50MB)

also graphed – a *with unpin* version where each page pinned for DMA is subsequently unpinned an a *no unpin* version. When unpinning as part of rendezvous, the dominating factor are the 6000us required to service an unpin request. The *no unpin* version is also included as many other high performance communications software determine that leaving memory pinned is a reasonable policy, which is obviously wrong when the amount of physical memory referenced over the lifetime of the program exceeds physical memory. Firehose excels in that the physical memory limitation is applies only to the active working set of pages and not all the pages touched during the entire run of the application. The *no unpin* version remains competitive with firehose until it maxes out on the available physical memory and crashes at M = 400MB. The reason rendezvous with *no unpin* can otherwise trail closely behind firehose because the remote node is fully attentive to the network and can also benefit from the overlap between bulk communication and pin request synchronization messages. The graph also shows that firehose degrades gracefully as the amount of reference memory pages passes M, then M+MAXVICTIM and eventually meets the rendezvous *with unpin* curve.

3.3 Firehose API

In order to simplify the porting effort of the firehose algorithm to the many flavors of pinning-based networking hardware, we factored out our initial implementation on Myrinet/GM into a full-fledged library. In investigating the requirements for a such an API, we discerned two types of pinning-based networks:

1. **Region-based**. The interface provided through the NIC's software and firmware can pin memory over regions of memory, which is a convenient way to associate a name to a given pinned memory region. Subsequent pinning/unpinning operations can simply use this name to enforce resource limits and even optimize for coalescing contiguous regions to create a new name. More importantly for the purpose of the research presented in this paper, region-based pinning does not suffer from as much resource contention as page-based pinning since the granule is a region and not an aggregate set of pages. The design and implementation of region-based pinning was carried out in another project parallel to this one and is the subject for different discussion.

2. **Page-based**. The underlying NIC's software interface does not provide any type of naming scheme for operations issued on regions – pinning operations apply exclusively to a set of pages. The side-effect of having anonymous pages as the granule for pinning and unpinning operations leads to higher contention, a condition that must be dealt with by firehose.

The full API exports enough functionality in order to make sure that firehose requests to pin local and remote memory are entirely handled by the firehose library itself. Clients can issue local and remote pin operations through distinct interfaces and each of these operations can be carried out synchronously by returning to the caller once completed or asynchronously by way of a callback provided by the client. Also provided are *try* versions of both local and remote pin calls, which allow non-binding requests in the case of a miss – hits however, will commit to the firehose table. This allows clients to be versatile in their use of firehose, such as issuing a one-sided operation if the remote pages happen to be pinned or resorting to a client-specific network out-of-band communication mechanism that doesn't require the page to be pinned.

Firehose, however, expects the client to provide active message-like functionality, such that requests and replies issued from firehose can be completed using the client's high performance network support. Since current clients are conduits part of the GASNet communications library, they already provide active message functionality as part of GASNet. Conversely, firehose clients must export the network specific call to issue the actual low-level library call to pin and unpin local regions of memory (in many cases, this call will be a wrapper around the NIC-specific software interface for registering memory regions).

Also, some other miscellaneous features have been added to the API. As part of the initialization function, clients can pass a list of regions that they have guaranteed to be pinned prior to firehose initialization. For example, some clients may wish to pin a portion of the program stack knowing that many one sided-operations will be issued from stack temporaries. The most important library calls are summarized in figure 3.3.

- `firehose_init(max_pinnable_memory,prepinned_regions,...)`
  At initialization, the client provides firehose with an upper bound on the amount of physical memory that can be pinned and can optionally pass regions that have been prepinned (such as the program stack).
- `firehose_local_pin(addr,len,...)`
  Pins a local region of memory and returns a descriptor of the region to the client.
- `firehose_try_local_pin(addr,len,...)`
  Immediate local pin operation, returns a non-NULL region descriptor if the region covered is entirely pinned. A successful operation always increments associated reference counts.
- `firehose_partial_local_pin(addr,len,...)`
  Returns the largest segment already pinned in the specified region.
- `firehose_remote_pin(node,addr,len,flags,callback,...)`
  Returns (synchronously or asynchronously) when the remote region is pinned, with optional flags such as enabling an optional callback at the target once the pin operation is completed.
- `firehose_try_remote_pin(node,addr,len,...)`
  Immediate remote pin operation, returns non-NULL region descriptor if the remote region covered is entirely pinned. A successful operation always increments associated reference counts.
- `firehose_partial_remote_pin(addr,len,...)`
  Returns the largest segment already pinned in the specified region.
- `firehose_release(...)`
  Allows firehose to release a previous local or remote request.

Figure 4: Some of the Firehose functionality available through the Firehose API

3.4 Maintaining Firehose Table Consistency

In a multithreaded environment, the number of firehose table users increases as the number of threads that are spawned, typically the number of host processors in a HPC environment. Firehose resources such as f and M+MAXVICTIM constitute a fixed set of resources allocated at each network endpoint (or node, for simplicity). In essence, participating threads must compete for this set of resources for every request made through the firehose API. Constraining firehose clients to remain within these parameters is a fundamental requirement of the algorithm – it is not a side-effect of being SMP aware. Actually, these parameters impose a limit on the amount and size of RDMA operations that can concurrently be put in flight, and as such require clients to poll the network to respect the firehose limits. Polling guarantees both that the limits on local pages and remote firehoses are respected by running active message handlers to encourage previously issued operations to complete.

Overall Requirements. Accesses to the firehose table require that the table return consistent value, namely that pages that are marked as pinned really are pinned and that misses in the table truly represent unpinned pages – both for local and remote pages. In the simplified firehose flow diagram for `firehose_remote_request()`shown in figure 2, a node accesses and updates the firehose table in both local instances when it satisfies a locally initiated remote request as well as in remote instances, when it satisfies a remotely initiated request in the context of an active message handler. Local and remote accesses to the table differ in that local accesses are fully synchronous while accesses as part of AM handler context are asynchronous. Both types of accesses for remote requests are explained in turn.

During a local access to the table, a client-initiated request assesses the pinned state of a set of remote pages and determines if the request can be classified as a full hit, or if it will miss. In the case of a hit (including a hit on a page in the remote FIFO), firehose will ensure that the referenced set of pages remain pinned until the client explicitly releases the pages (through `firehose_release()`) by incrementing a reference count for each remote page. If the request misses, the referenced pages can only be marked as pinned when the pin operation has actually completed. However, in accordance to our design goals of reducing the amount of pinning synchronization messages, we've introduced a PENDING state which allows subsequent requests that happen to reference exclusively pages that are known to be pending to simply defer a local callback instead of sending an additional active message. The second access to the table happens once the RDMA has completed and the client calls `firehose_release()`. In this case, the reference counts associated to pages to be released are decremented and 0-refcounts are moved into the remote per-node FIFO.

Accesses to the table as part of a remote request is only executed from within an AM handler. Particular AM semantics specify that handlers cannot poll, block or spin-poll for an unbounded amount of time [6]. In order to allow table updates (and any accompanying pin/unpin operations) to be completed from AM handler context, all accesses to the table must be protected by locking the table for a deterministic amount of time. Since client RDMA operations are typically split phase in their initiation and completion, the table consistency must be guaranteed throughout each split-phase table access (whether the access be from AM handler context or client initiated).

Steady State Requirements. At steady state, firehose state tables resemble figure 1, where all of the alloted f firehoses are mapped to remote pages. Future requests will have to move an equal amount of stale firehoses as the request requires in new firehoses. As the number of RDMA requests (and/or the size of these requests) increase, the number of requests that can be sent becomes proportional to the speed at which stale mappings are recovered and eventually used as replacement firehoses. From a single client request's point of view, recovering replacement firehoses to satisfy the f requirement entails polling the network in order to run the completion handlers of previously issued remote requests. This polling is precisely the focus of the difficulties in dealing with more than one firehose client per single firehose state table as available resources must be correctly and fairly partitioned.

4 SMP-aware Firehose

The design of an SMP-aware firehose algorithm necessitates a reevaluation of the many assumptions made in the original non-SMP design. Since accesses and updates to the firehose table initiated through the firehose API originate from more than a single client thread, firehose polling at steady state must be modified to provide two major goals:

- **Correctness**. The implementation must allow a consistent view of the local and firehose tables throughout all the possible states of a firehose request. This requirement applies to table accesses initiated from competing client threads as well as from within active message handlers.
- **Fairness**. Competing threads should be provided with a mechanism that allows each individual thread to make forward progress in both its requests for local and remote pages to be pinned.

As explained in section 3.4, interesting conditions for both SMP and non-SMP firehose clients arise at steady state, when `firehose_remote_request()`must poll for replacement firehoses. Two types of resources must be respected with firehose:

1. **Per-node firehoses** f. Each node can map up to f firehoses to each other node, where a single firehose is equal to a memory page mapping. Clients request an amount in firehoses equivalent to the size of the RDMA operation to be initiated at the target remote memory window. According to the sizes and amount of the RDMA operations being concurrently initiated by competing client threads, only a fraction of these f firehoses are available. In fact, client threads can livelock when all available f end up being distributed among competing threads without any client RDMA operation being satisfied in its individual need for firehoses.
2. M+MAXVICTIM **local pages**. Each node has a hard limit of M+MAXVICTIM local pages being pinned, as established by the physical memory limitations of the underlying system. As it is the case for f, threads also compete for this local resource and can lead to livelock.

For example, a problem case can be envisioned for `firehose_remote_request()`when, say, $f = 10$ and two client threads request that 8 new firehoses be mapped. One thread could obtain the first available 5 firehoses from the pool of unused firehoses while the second obtains the next 5 by recovering firehoses through polling of recently completed operations. Although most of this paper is focused mostly on the SMP-awareness of remote requests, local pin requests are also submitted to resource limitations that require special attention. Consider a case where threads concurrently pin large amounts of local memory as per the `firehose_local_request()`client request. In order to remain within the M+MAXVICTIM hard limit, threads poll for replacement local firehoses and can lead to deadlock if none of the thread requests are able to make progress.

4.1 Firehose SMP Design

The existing approach for satisfying a remote request is simple – separate the pinned pages from the unpinned pages and poll until an equal amount of replacement firehoses can be used to substitute the set of pages to be pinned. If there aren't enough available replacements in the FIFO, the calling thread polls the network until enough replacements can be recovered. For reasons explained above, livelock may occur when competing threads unfortunately split the amount of available replacement firehoses, which eventually leads to deadlock when there are no replacement firehoses to be obtained. The mechanism proposed resolves deadlock in an optimistically concurrent manner – competing threads poll for a specified amount of iterations before declaring deadlock. In all instances of the protocol described below for acquire remote firehoses, deadlock avoidance is applied on a per-node basis – threads compete only if they concurrently target the same node.

This approach can be contrasted to a purely pessimistic approach, one where progress for recovering replacement firehoses would be serialized such that at most a single thread is polling at one time. While concurrent threads issuing requests large enough to be a significant fraction of f constitutes the case that leads to serialization, we expect most requests to be reasonable in the ratio of requested:available firehoses. Another design opportunity could have been based on allowing threads to issue requests in fragments, such that a full remote pin request is satisfied by a series of smaller pin requests that could be issued as soon as replacement firehoses would be available. This opportunity does not warrant further research as it seriously impacts the throughput of client requests and puts a damper on our goals of keeping two-sided host synchronizations at a minimum.

```
my_da = 0;                  /* Deadlock avoidance counter */
while (my_da < DA_LIMIT)
{
    my_da++;                /* increment DA bit */

    if (fh_all_da[node])    /* Someone else won the deadlock avoidance.
      fhsmp_Rollback(...);   * Return what we got from the fifo and remove
                             * new entries we created */

    UNLOCK_TABLE;           /* Leave the table in a consistent state */
    AM_Poll();
    LOCK_TABLE;
    avail = (f - f_used[node]) + fifo[node];

    if (avail == 0)
        continue;

    /* If new replacements, reap them and mark them as PENDING_UNCOMMITTED */
    last_addr =
         fhsmp_PinWithLogAgain(avail,last_addr,...);

    if (last_addr == end_addr) /* Commit. Each new entry changes from
        fhsmp_Commit(...);      * PENDING to PENDING_UNCOMMITTED, and
                                * send the Active Message Pin request */
}
/* At this point we proceed knowing we will eventually make progress */
```

Figure 5: Deadlock avoidance protocol for remote Firehose requests

Remote Client Requests. Figure 5 shows a simplified version of the work to be done before declaring deadlock. A given thread will try to recover firehoses for up to `DA_LIMIT` attempts and declares deadlock if another thread hasn't done so. The approach is optimistic in the common case, which implies that a given thread should be able to reap enough replacement firehoses to satisfy its request without declaring deadlock. Until any thread has posted the deadlock avoidance bit `da`, all threads try to make progress with the replacement firehoses they can recover and set any new firehose as PENDING_UNCOMMITTED. The introduction of this new firehose state was required for the SMP-aware code as threads must poll without knowing if the progress they have made so far in acquiring replacement and creating new firehoses will be able to commit. After each poll to service the active message queue, a thread checks to see if another thread has posted the `da` bit and aborts it's request if so. Aborting is similar to a rollback-like process, where all the replacement firehoses are returned to the FIFO and all the newly created PENDING_UNCOMMITTED firehoses are destroyed. Following a rollback, a thread suspends itself and waits for a condition variable corresponding to a "no deadlock" condition to be posted by the winning thread at which time the entire operation is restarted.

The proposed approach to deadlock avoidance provides both of the characteristics required in SMP-aware firehose. Correctness is handled individually by each thread by not exceeding any of the per-node firehoses and the newly introduced PENDING_UNCOMMITTED state is handled correctly in various areas of the firehose code. Fairness is provided in theory by allowing each thread to make an equal amount of attempts before deadlock (experimental fairness is reported in section 5).

Given that common case operation should not lead to deadlock, the above deadlock avoidance protocol should only be entered during times where deadlock is possible. If it can be determined early in the request that there are enough replacement or unused firehoses to satisfy the operation, an easier path can be taken since we are guaranteed to commit. This "easy path" is shown in the flow diagram of figure 6, where a process called Estimate Remote

Request analyzes the client request to see if it can be satisfied entirely without polling – either the request is a perfect hit or a a miss can find enough replacements in what's currently available.

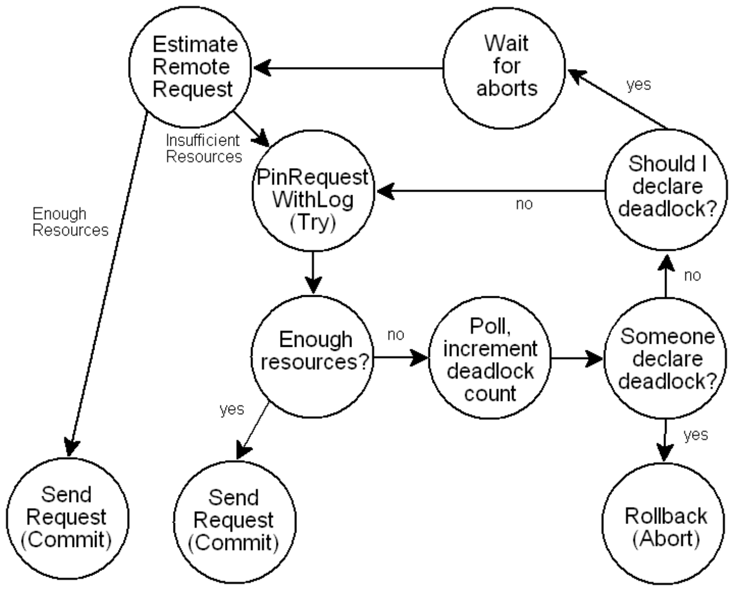

Figure 6: Flow diagram for Firehose's optimistic deadlock avoidance protocol

Local Client Requests. As is the case for remote requests issued through `firehose_remote_request()`, `firehose_local_request()`requests implement a similar a similar but simpler version of the deadlock avoidance protocol. Although polling is also required to make progress on previously issued local pin requests and eventually recover unused page mappings, there is no remote messaging involved in pinning local pages. However, clients are equally susceptible to hit problem cases in the presence of a large number of requests. For example, clients that wish to issue many *get* operations in that follow an irregular access pattern will cause an increasing number of local requests to miss in the firehose table and as such will increase the need for replacement firehoses. The local deadlock avoidance protocol is similar to the remote one, except that condition variables are used more aggressively to minimize the latency between threads that have suspended after rolling back and threads that have won the DA race and can complete their requests.

There are many other subtleties that are overlooked in the simplified pseudocode of figure 5 and omitted from the presented design issues for local pin requests, but these are extraneous to the core discussion and are summarized in table 3.

5 Results

This section presents results obtained with the SMP version of page-based firehose implemented for this paper, which composes 1400 of the 6500 lines of code for the full firehose library (regrouping both page and region based firehose implementations). Firehose's SMP-awareness thus constitutes an important fraction of the library and the results in this section attempt to validate both the correctness and fairness of Firehose-SMP. The code is not as mature as the non SMP-aware firehose version and has not been tuned yet. Since the primary goal is to provide correctness and fairness, something the previous code could not handle, tuning and benchmarking the firehose SMP implementation is slated as future work. Before reporting any of the numbers below, we testing our implementation against some applications and kernels part of the UPC benchmark to verify overall correctness and have generated synthetic deadlock-prone communication traces to make sure that every code path firehose SMP deadlock avoidance and recovery is functional.

In order to validate our claim that there is merit in sharing the firehose table amongst more than one computation thread, we ran the GUPS UPC application benchmark, which is really meant to stress and measure the performance

Component	*Problem*	*Solution*
Deadlock Avoidance	Competing threads can hit on a firehose that was used as a replacement before polling	After each poll, assert that all replacement firehoses were untouched and rollback if not
Estimating Remote Requests	Hitting on a PENDING_UNCOMMITTED firehose during estimation would not create a reliable estimation of required resources	If the thread won the DA, poll to allow losing threads to remove their PENDING_UNCOMMITTED firehoses. If not, increment the local DA count and rollback.
Estimating Local Request	Hitting on a PENDING local page cannot count for a hit	The local pin is handled in two phases, and the PENDING status is cleared only after the page is pinned. All local requests that hit a PENDING page must rollback
AM Handler Pinning	Requests to pin local memory hit a PENDING page and AM handlers can't poll or rollback	Create a special queue for these requests and drain it after every pin operation
Firehose replacement	Before a node reaches steady-state, no firehoses are mapped and available for replacement	Firehose keeps a count associated to the number of firehoses used and can allow a replacement to be satisfied by an unused mapping (new firehose needs to be moved).

Table 3: Summary of some of the corner cases that appear in maintaining consistency throughout multi-threaded split-phase firehose table accesses

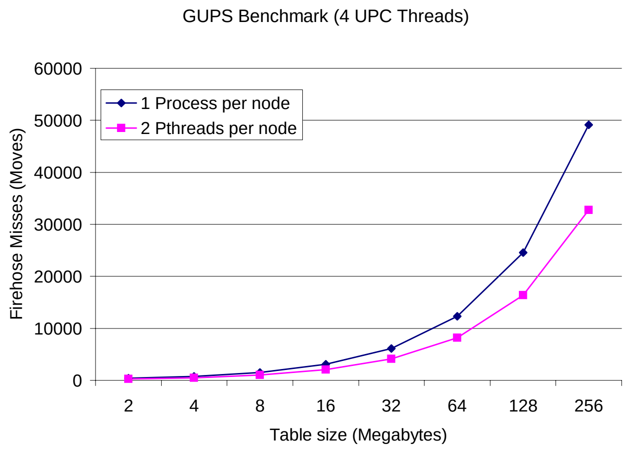

Figure 7: UPC GUPS application benchmark, shows that sharing the firehose table reduces the amount of firehose misses

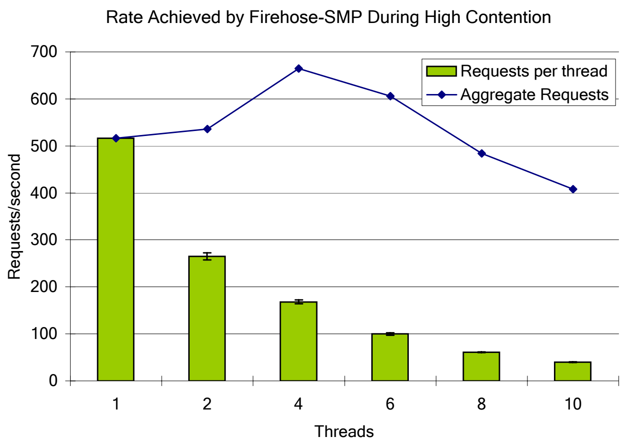

Figure 8: 32Kb Put rate (both aggregate and per thread) achieved using firehose-SMP as the number of competing client threads increases. Error bars for 2 threads or more represent the rate obtained by competing threads.

of high performance memory systems. While GUPS is not an ideal candidate to measure the performance and even correctness of firehose SMP, its random 8-byte updates to various areas of the UPC global shared heap provide interesting communication patterns. These updates translate into firehose requests to pin memory referenced by all participating computation threads, which means that two UPC computation threads per node on the dual-processor test system share a single firehose table. The graph compares the number of firehose moves requested during the entire application run between two computation thread layouts – one where 4 UPC threads are mapped to single processes on four nodes and another where 2 Pthreads are run within a single process on two nodes. As we increase the GUPS table size, the number of firehoses misses generated by each node decreases (with a 35% improvement in the case of a 256MB table). As such, firehose-SMP does help in reducing the number of firehoses misses in a CLUMPs configuration.

Testing the implementation for correctness is a much more difficult task. Since most of the problem cases are difficult to predict, let alone to generate from the point of view of an application, synthetic benchmarks were created to validate the correctness of the algorithm and test its implementation. For example, reducing the M and M+MAXVICTIM parameters and issuing remote put requests very close to the limits can synthetically generate high contention. We've adapted some of the GASNet benchmarks to spawn large amounts of threads and test every aspect of the implementation – local and remote rollbacks when other threads acquire the deadlock avoidance bit, correct handling of firehoses in pending states, etc. As such, we are satisfied with the level of correctness currently implemented in Firehose SMP.

The fairness property was tested by using a benchmark that scatters 32Kb puts from the same source location but to a random location at a large remote memory space. Since 32Kb is 8 memory pages, we assigned f to 8 such that only 8 firehoses be available to all threads – this requires each requesting node to move existing firehoses in the case of a miss (which is the common case on a large target memory space). In figure 8 we report the rate at which firehose requests could be issued by each competing client thread as the number of client threads are increased (we report the obtained median values out of 10 runs). It should also be noted that since the test system has only two processors, testing with greater than four threads does not represent a case that will be replicated by firehose clients. Similarly, we do not report values for an odd number of threads as we want each processor to be assigned an equal amount of runnable threads. It should be noted that in all cases, the deadlock avoidance path was taken between 75% and 85% of the time depending on the number of threads, which confirms that we are effectively testing fairness with and without deadlock avoidance being used. A few fairness guarantees can be validated from this graph:

- The error bars are plotted for thread counts greater than 1 to represent the rate obtained by all competing threads. Very small errors show that all competing threads post a similar rates.
- The aggregate rate of all thread requests is also plotted as a line on the graph and the fact that it slowly decreases shows that the implementation does not suffer any important performance degradation in particular parts of the code. Rather, the decreasing aggregate rate is a sign of increasing the amount of work for the thread scheduler. The peak at around 4 threads is a side-effect of keeping each processor busy with at least two instead of one runnable communication bound thread.

Among other points of interest is the rate at which deadlock recovery is resorted to as the available number of firehoses f is increased. Additionally, the actual application speedups allowed by sharing the firehose table are also important to firehose clients. However, this the requirement for bug-free operation constitutes the major hurdles in the implementation, we leave performance analysis as an upcoming research topic.

6 Conclusion

One-sided zero-copy RDMA communication is the preferred approach for obtaining high-performance and low-bandwidth transfers on high-performance networks. While many techniques have been proposed in the past to optimize for high bandwidth transfers, the true problem of not burdening applications with explicitly pinning remote pages for RDMA had never been dealt with entirely. With the Firehose algorithm, clients benefit from one-sided zero-copy RDMA in the common case and pay some synchronization overhead in the uncommon case. The original firehose paper [1] showed that in two application kernels (Cannon Matrix Multiply and Parallel Bitonic sort), up to 99.8% of all communication could be completed entirely one-sided.

This paper proposes mechanisms to handle sharing of a common firehose table between two or more computation threads. On a non-SMP configuration, the firehose table access patterns cause split-phase accesses and updates to the table in the context of both updating table entries for locally and remotely initiated pinning operations. With threaded CLUMP clients, split phase accesses to the table turn into concurrent split-phase accesses, which leads to resource partitioning problems and complications in maintaining firehose table consistency. Using an optimistic method of recovering replacement firehoses, we allow each thread to make independent progress in the case of low contention and resort to a deadlock avoidance protocol in the uncommon case of high contention. In all areas of the presented analysis for an SMP-aware firehose implementation, worse-case operation is considered and low overhead solutions are presented even though common case operation should not require deadlock avoidance. For these uncommon cases, deadlock-prone communication traces are used to verify and validate that the implementation is working correctly and remains fair with respect to all competing threads.

References

[1] C. Bell and D. Bonachea. A new dma registration strategy for pinning-based high performance networks. In *Workshop on Communication Architecture for Clusters (CAC'03)*, Nice, France, April 2003.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[3] D. Bonachea. GASNet specification, v1.1. Tech Report UCB/CSD-02-1207, U.C. Berkeley, October 2002.

[4] J. Nieplocha, V. Tipparaju, and D. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. In *Workshop Communication Architecture for Clusters (CAC02) of IPDPS'02, Ft Lauderdale, FL*, 2002.

[5] Quadrics Supercomputing. *Quadrics QSNet Interconnect*, 2002. http://www.quadrics.com.

[6] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.