A New DMA Registration Strategy for Pinning-Based High Performance Networks

Christian Bell Dan Bonachea
Computer Science Division
University of California, Berkeley
Berkeley, California, USA

E-mail: {csbell,bonachea}@cs.berkeley.edu

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

Abstract

This paper proposes a new memory registration strategy for supporting Remote DMA (RDMA) operations over pinning-based networks, as existing approaches are insufficient for efficiently implementing Global Address Space (GAS) languages. Although existing approaches often maximize bandwidth, they require levels of synchronization that discourage one-sided communication, and can have significant latency costs for small messages. The proposed Firehose algorithm attempts to expose one-sided, zero-copy communication as a common case, while minimizing the number of host-level synchronizations required to support remote memory operations. The basic idea is to reap the performance benefits of a Pin-Everything approach in the common case (without the drawbacks) and revert to a Rendezvous-based approach to handle the uncommon case. In all cases, the algorithm attempts to amortize the cost of synchronization and pinning over multiple remote memory operations, improving performance over Rendezvous by avoiding many handshaking messages and the cost of re-pinning recently used pages. Performance results are presented which demonstrate that the cost of two-sided handshaking and memory registration is negligible when the set of remotely referenced memory pages on a given node is smaller than the physical memory (where the entire working set can remain pinned), and for applications with larger working sets the performance degrades gracefully and consistently outperforms conventional approaches.

1. Introduction

Many high performance networks leverage user-level RDMA as a means of achieving high bandwidth transfers. Research has shown that it is beneficial to allow zero-copy RDMA operations to be initiated from user-level, given that memory protection is suitably implemented between the OS and the network interface. These architectural trends, implemented by relatively low-cost, high performance network interconnects [9, 24], have helped in bringing traditional highly-integrated systems to the commodity market. These systems typically support a version of the Message-Passing Interface (MPI) [6] through their own low-level messaging libraries, which offer limited (Myrinet) to extensive (Quadrics) hardware support for remote memory operations. The emergence of MPI as the primary parallel programming paradigm can be explained by the mutually profiting relationship it has established with commodity network interconnects. Conversely, Global-Address Space (GAS) languages (such as Titanium [29], UPC [14] [1] and Co-Array Fortran [21]) have not traditionally received the same level of attention across various high performance computing manufacturers.

Among other things, GAS languages differ from traditional message-passing interfaces by promoting the programmability of shared-memory systems while still providing the performance and control over data layout available through message-passing systems. While this model maps very naturally to tightly coupled SMPs, the shared-memory model tends to expose design restrictions of some commodity networks – particularly in their use of DMA operations on remote memory locations. In fact, some networks (hereafter "pinning-based" networks) require all memory which is to be made accessible for remote DMA to be explicitly locked (pinned) by the target host software before the transfer can proceed. DMA registration, (or the combined cost of pinning and unpinning DMA pages) can constitute a significant performance bottleneck on networks such as Myrinet. GAS language applications tend to be more susceptible to these problems than those written in a message-passing style, because a large fraction of the application's virtual memory space may potentially be accessed by one-sided remote memory operations.

The following paper demonstrates a new memory registration algorithm for pinning-based networks which is general enough to satisfy the most demanding RDMA requirements, and compares it against existing registration strategies for pinning-based NICs.

The Firehose algorithm is implemented using the GM lightweight messaging interface on top of Myrinet [18], which is currently the most popular pinning-based high performance network used for building commodity HPC clusters. Additionally, this work is part of an implementation of the GASNet interface [10], a portable lightweight communication interface used for implementing several SPMD GAS languages.

Section 2 provides background information about DMA registration. Section 3 explains the requirements of GAS languages,

presents the Firehose algorithm itself, and discusses how it compares with existing DMA registration strategies. Section 4 provides performance results for Firehose on synthetic benchmarks and real applications. Section 5 discusses related work on DMA registration, and we conclude in Section 6.

2. Background

Network DMA interfaces can be divided into two broad categories based on their memory registration support, either automatic hardware-assisted registration or passive pinning-based registration. With a hardware-assisted approach, the client software is relieved from explicitly managing and pinning memory to be used for DMA and NIC hardware is used to perform registration on-demand.

The hardware-assisted 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 allow the NIC to pin pages, initiate page faults when necessary and track changes in the application's page table (see Quadrics [24]). 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. Remote put/get operations may be initiated from anywhere in the user's virtual memory space and the network hardware guarantees delivery given that the source and destinations map to existing pages in the application's page table. Pushing the responsibility of enabling pages for DMA down to the network hardware comes at an expected increase in hardware cost and complexity, but also leads to significant software maintenance costs. On top of requiring platform and OS-specific modifications, hardware-assisted approaches must generally be continuously updated as changes are made to internal Virtual Memory subsystems (which is likely to occur frequently in open source OS's). Nevertheless, in terms of performance and scalability, the advantages of these networks usually outweigh the disadvantages, which makes hardware-assisted registration the preferred approach for GAS language implementations.

The second general registration approach, pinning-based, 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 swapped 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). Some NICs (such as Infiniband) may also have limits on the number of separate regions of contiguous virtual pages that can be simultaneously pinned.

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, [20] 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. This paper is concerned with an additional metric, the ability to provide entirely one-sided remote memory operations as a common case. GAS languages excel in their ability to overlap communication with computation and other communication through compiler analysis and fine-grained communication scheduling, however such optimizations are most effective with communication systems that provide low-latency, one-sided remote operations.

3. Firehose Algorithm

The proposed Firehose algorithm seeks to provide one-sided operations as a common case to reap the benefits of faster response time and to avoid interrupting remote host processors. The algorithm also promotes zero-copy operations to lower CPU overhead and allow overlap with computation and other communication. Although the costs of DMA registration cannot be eliminated altogether, Firehose attempts to amortize these costs over many operations while allowing one-sided operations as a common case. The algorithm is presented through the next three subsections: requirements of GAS languages that motivated the need for Firehose, a description of the algorithm itself, followed by a comparison with other DMA registration approaches.

3.1. GAS Shared-Memory Requirements

The message-passing paradigm provides good performance for parallel applications that can be cast into a bulk synchronous communication pattern (mainly through the ubiquitous MPI), however it lacks the programmability of shared-memory style programming. GAS languages attempt to consolidate both approaches, both on traditional highly-coupled systems and distributed systems such as networks of workstations (clusters). The global shared memory abstraction provided by GAS languages makes it possible to program parallel applications in a shared-memory style, regardless of how the memory is organized in the underlying hardware, while still providing good performance and control over data placement. While high-performance networks for commodity cluster computing have been performing rather well in the realm of message-passing over the years, many still do not have good support for implementing globally-shared memory. The limiting factors are strongly related to their support for low-latency, low-overhead one-sided remote memory operations or in the amount of memory that can be made available for remote operations.

GAS languages encourage the use of distributed data structures – for example, UPC provides language-level support for shared arrays striped across nodes. As this translates into providing shared

memory across disjoint physical memory spaces, local memory references and computation are frequently interleaved with remote memory references through the network interconnect. GAS languages also make it easy to express remote memory operations, and consequently the performance of GAS language applications tends to be sensitive to the latency and CPU overhead associated with performing small (generally $\leq$ 8 byte) remote memory operations. As a result, GAS language implementations are generally carefully designed to support low-latency, low-overhead, small remote memory operations, in addition to the traditional design goal of providing high bandwidth for large message data transfers.

Using a communication system such as GASNet, language-level remote memory references are translated into network communication events – typically one-sided gets and puts. Consequently, the availability of truly one-sided remote memory operations is important for the efficient implementation of GAS languages – specifically the ability to perform puts or gets on remote memory without interrupting the remote host processor or waiting for it to explicitly poll the network. In [7], we extract the performance parameters from several high-performance networks through microbenchmarks and evaluate the level of support for one-sided operations, with the goal of using this information to guide communication scheduling optimizations over these one-sided operations in GAS language compilers.

GAS language applications are prone to use large data working sets, which has an important influence on the expected communication and memory access patterns. For example, large distributed shared memories make it easier for programmers to use unstructured distributed meshes (rather than structured ones) and distributed sparse matrices (over dense ones) – algorithms over such irregular data structures are generally more difficult to express using explicit message-passing communication. Although it would be difficult to come up with a generalization relative to the size of the working data sets of such algorithms, it can be maintained that memory references (or memory access patterns) are prone to be directed over a large portion of memory, possibly bounded only by virtual memory. Since this paper is mainly concerned with making it possible to efficiently support large amounts of shared memory over clusters, the following points enumerate the important parameters in evaluating a memory registration strategy for implementing GAS languages over pinning-based networks:

1. **Memory usage and size of working set**
   Memory usage represents the total memory that is used throughout the entire run of the application. If this value is within reasonable limits of the amount of physical memory, it may be reasonable to simply pin everything at startup. This would enable one-sided DMA on every remote operation initiated from or to the pinned memory region. Conversely, the size of the working set represents the data structures and any other program code being actively accessed over some appropriate time period. The working set is likely to be limited by the amount of physical memory (otherwise the application would frequently be swapping), and this tendency can be exploited in designing an adaptive memory registration strategy.

2. **Memory access pattern**
   The pattern of memory allocation and access on remote nodes affects the way pinning behavior evolves within the application memory space. GAS languages differ in allocation patterns. For example, the current implementation of Titanium has no specific allocation pattern, such that remote memory accesses may appear scattered in virtual memory. Conversely, UPC implementations generally manage remote DMA segments as heaps and follow a well-defined memory allocation pattern: heaps grow monotonically upwards or downwards in response to the application's memory requirements [13]. (Additionally, multithreaded UPC implementations divide a large segment into smaller thread-specific memory heaps). This regularity can be exploited in designing a memory registration strategy.

3. **Cost of registration**
   Registration cost, specifically the time required to pin and unpin a memory page on the given network interconnect, can be a significant performance factor. This paper defines registration cost as the combined cost of the system calls and NIC-specific commands necessary to pin pages into memory and subsequently unpin them. Techniques for lowering the overhead of a single registration operation are generally system-specific and beyond the scope of this paper – we instead focus on strategies to reduce the frequency of these registration operations.

4. **Cost of synchronization messages**
   Some registration strategies require nodes to exchange synchronization messages to accomplish registration (for example, Rendezvous requires a round-trip of synchronization messages to register memory before the transfer begins, and some final synchronization messages to deregister after the transfer). The frequency and cost of these messages affect application performance, both by consuming bandwidth and standing in the critical path of remote accesses, thereby increasing remote access latency. The latency of synchronization messages which require a response may also be affected by the remote host's attentiveness to the network. An efficient registration strategy should strive to reduce the number synchronization messages necessary.

3.2. Algorithm Description

The Firehose algorithm starts by determining the largest amount of application memory that can be registered. 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.

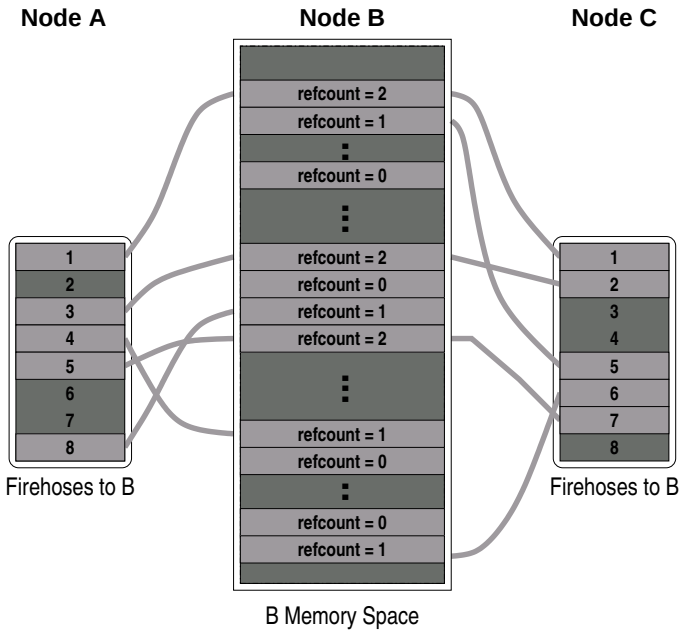

Figure 1: Runtime snapshot of two nodes (A and C) mapping their firehoses to another node (B)

Implementation of the Firehose algorithm requires a thin control layer (such as Active Messages [27]) for the handshaking that takes place when a host wishes to move a firehose [1]. The following steps illustrate how a remote memory put operation can be completed with Firehose:

1. The put operation 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; if not, the second step follows.

2. A firehose move 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 operation may be sent.

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:

- Virtual pages may be grouped together into contiguous multi-page "buckets" (with a size fixed at compile-time) which are always managed together to effectively increase the page size of the system and reduce the size of Firehose bookkeeping data. The use of multi-page buckets may also help to limit the number of separately pinned regions of pages, on networks such as Infiniband where such regions may be a limited resource;

- Each node associates a reference count with every locally-pinned bucket to track usage. While the reference count is greater than zero, this bucket is not a candidate for unpinning because it is currently in-use by one or more incoming firehoses or locally-initiated operations. While the bucket is pinned, subsequent remote requests to attach a firehose to this bucket or locally-initiated operations needing this bucket 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 bucket, 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 bucket-based reference count. By allowing a configurable number of 0-refcount buckets 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 bucket 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 buckets with zero reference counts. Victim buckets are evicted when firehose move requests arrive for buckets 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. Updating the count constitutes a fairly low level of overhead and synchronization, which turns out to be practical for multi-threaded implementations of Firehose.

[1] Active Messages are an integral part of the GASNet API, and the synchronization messages in our Firehose implementation use the same AM-over-GM framework discussed in section 4.1

Table 1 summarizes the data structures used to implement the Firehose algorithm with all these optimizations.

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

Table 1: Primary data structures in Firehose implementation

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

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 bucket 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 bucket reference count reaches 0, it is added to the head of the victim FIFO queue - when the queue length exceeds this configurable parameter, buckets are removed from the tail of the queue and unpinned.

3. **Bucket size**
   Although Virtual Memory Managers use page size as the basic unit of virtual and physical memory, the Firehose algorithm deals with memory in units of bucket size. Bucket size is equal to page size by default but may be configured to be a multiple of page size to adapt to other requirements (for example to reduce the size of the Firehose bookeeping data). Note that allowing buckets to span more than a single page may present new challenges since bucket registration requests may cross page boundaries into unmapped virtual space (causing segmentation faults) - some memory allocator support may be required to allow multi-page buckets.

In our implementation, bucket size is set at system compile time, and M and MAXVICTIM are set at runtime by the user (or more likely the person installing/tuning GASNet for a particular site). 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, benefitting 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).

A future work item is to make the victim FIFO length limit adaptive – i.e. allow the victim FIFO to grow beyond MAXVICTIM (perhaps up to some higher upper limit) provided the total amount of pinned memory is still below the M+MAXVICTIM hard limit. This optimizes for the case when the page demands of remote nodes are unbalanced – it allows us to keep extra pages in the victim FIFO to optimize the performance of the remote nodes who are targeting this node heavily, essentially allowing them to "borrow" some of the physical pages which were statically allocated to other remote nodes that are not using all their firehose connections to this node. This also helps to ameliorate potential scalability problems in the static allocation of firehose resources by reducing the performance penalty in an unbalanced communication pattern when a remote node has insufficient firehoses to cover its working set (most of the unpin-repin costs can still be avoided).

3.3. Firehose and Existing Pinning-based DMA strategies

Although hardware-assisted memory registration eases the porting task for GAS languages, most existing high performance NICs are not equipped with the sophisticated memory interface hardware required to make this approach work. Table 2 provides a summary of current DMA registration strategies, separating hardware-assisted from software-based approaches. Their advantages and disadvantages are explained below.

Of the pinning-based DMA strategies listed in the table, the Pin Everything approach is different since pinning is not done on demand – a single segment of memory is pinned at startup (or equivalently, pages are pinned as they are first used by the memory allocator and kept pinned for the life of the program). If total memory requirements are known to be constrained to a reasonable size within the physical memory limits of the host, it might be preferable to preemptively pin the entire remotely-accessible region of memory at startup in this manner. In this case, every DMA operation falling within the segment can complete as one-sided since it does not require additional synchronization between hosts to complete.

Strategy	*Advantages*	*Disadvantages*
Hardware-assisted	Zero-copy, One-sided, Full memory space accessible, No handshaking or bookkeeping in software	Hardware complexity and price, Kernel modifications
Pin Everything	Zero-copy, One-sided (no handshaking)	Limited memory, May require a custom memory allocator
Bounce Buffers	No registration cost at runtime, Full memory space accessible	Two-sided, Local copy costs (CPU consumption), Messaging overhead (metadata and handshaking protocol)
Rendezvous	Zero-copy, Full memory space accessible, Only handshaking is synchronous	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)

Table 2: Summary of available DMA registration strategies

Once the memory region is pinned, it is used without interruption (pinning/unpinning) throughout the entire run of the application. Otherwise unusable with larger memory requirements, systems using the Pin Everything approach will typically provide a special memory allocator that manages the pool of pinned pages and requires applications to allocate all remotely-accessible memory using this allocator. This approach may still incur some on-demand registration costs for pinning local memory if the client is permitted to initiate operations to/from local memory areas outside the pre-pinned segment (or alternately, bounce buffers could be used for local memory, as described below).

For systems with larger memory requirements, some level of synchronization is required between hosts in order to permit remote access on demand.

The Bounce Buffer approach uses temporary buffers residing in pinned memory to hold data for outgoing or incoming DMA operations. These buffers are posted based on criteria such as message sizes or fair sharing of buffers between nodes. Once a DMA operation completes in the case of a put, the target processor is informed of its delivery and must copy the data to the final remote destination. Similarly, when a get request is received, the targeted node copies the data into a bounce buffer and effectively executes a put operation to the requesting node. The main advantage of using bounce buffers lies in the cost of registration being paid only at startup – once the bounce buffers are pinned, no additional memory regions need to be pinned and all the user's virtual memory space is effectively accessible for puts/gets. The primary disadvantage is that the Bounce Buffer approach is strictly two-sided, and consequently the latency for remote operations is likely to suffer. Additionally, copying costs may be significant (even for small messages) since they interrupt remote host processors and dirty CPU caches and TLB's, and the CPU overhead for local copying restricts the potential for computational overlap (especially with large messages). Moreover, the task of managing the bounce buffers while still providing good communication concurrency and scalability can introduce significant complexity and handshaking overheads, depending on the level of hardware support. Finally, the approach remains questionable in the presence of highly-tuned communication systems where a wide range of non-blocking operations and fully-threaded client configurations increase concurrency of communication. There is a scalability problem when the number or size of network operations does not match the anticipated load, as bounce buffer efficiency relies on the rate at which buffers are posted relative to the rate of incoming operations.

For large messages, the cost of pinning on demand using a Rendezvous protocol can be amortized over more data and provide a net performance improvement over the use of many bounce buffers. Rendezvous protocols are carried out in two steps. The first sends a message to the remote node indicating the region to be pinned for DMA access. For puts, the remote node processes the message and pins the relevant memory region, then sends a reply to indicate the DMA operation can be initiated. A similar approach may be taken for gets, although as an optimization the reply may coalesce acknowledgment and payload (if permitted by the underlying network hardware). Optionally, there may be some final handshaking to unpin the relevant regions once the DMA transfer is complete. Even though both hosts must initially be synchronized, Rendezvous only requires the handshaking phase to be synchronous (and not the entire data transfer). Once the local and remote pages are pinned, or known to be pinned, the requesting node is free to initiate any DMA operation without interrupting the host processor. However, the cost of registration is paid on every operation, which is prohibitively costly for small messages and debatable for larger messages. Nieplocha et al. [20] show that the message size crossover point for achieving optimal bandwidth (using Bounce Buffers for small messages and Rendezvous for large messages) is greatly dependent on the underlying network hardware and software. On Myrinet/GM, the cost of unpinning constitutes a very large portion of the total registration cost. For this reason, most existing Myrinet/GM communication layers use Rendezvous and omit the unpin step (e.g. MPICH-GM [4]) or simply discard Rendezvous altogether in favor of two-sided pipelined Bounce Buffers [19]. Fortunately, registration cost is network dependent – previous work [20] has shown that other networks such as Giganet/Emulex cLAN offer more competitive memory registration performance, although the cost is still significant.

4. Results

In order to demonstrate the Firehose algorithm, we've developed an initial implementation and measured its performance on two application benchmarks and a synthetic microbenchmark written in Titanium. These were run on the LBNL Alvarez cluster, which consists of 2-way PIII-866Mhz nodes with 1GB of RAM and Myricom's Myrinet 2000 PCI-C network interface cards.

Titanium[15] is a Global-Address Space language and an ideal

client for the Firehose algorithm. Titanium is an explicitly parallel SPMD dialect of Java developed at UC Berkeley[5] to support high-performance scientific computing on large-scale multiprocessors, including massively parallel supercomputers and distributed-memory clusters with one or more processors per node. In Titanium, all data has a user-controllable processor affinity, but parallel processes may directly reference each other's memory to read and write values or arrange for bulk data transfers.

Titanium's distributed memory backends can utilize a wide variety of high-performance network interconnects, including Active Messages-2 [17], MPI, IBM LAPI, Cray shmem, and Ethernet/UDP. We've recently added support for using our GASNet [10] implementation (which uses the Firehose algorithm on Myrinet networks) as the communication system.

4.1. Platform description: GASNet and GM

GASNet provides a two-tier interface: a core and extended API. The core is based on the Active Messages [27] paradigm and is general enough to also implement the extended API. The extended API provides higher-level operations such as blocking and non-blocking puts and gets (with a rich set of synchronization options) and barriers. GASNet can quickly be implemented on a new network platform simply by implementing the core API and using a default reference implementation of the extended API written in terms of the core. Although this port is sufficient, it is often sub-optimal: implementors are encouraged to directly implement selected operations in the extended API to take advantage of any hardware support or special features provided by the given network (for example, remote DMA or hardware-assisted barrier). In any case, the GASNet interface is flexible, expressive and capable of efficiently implementing medium and high-level operations for many remote memory and collective operations. Figure 2 shows how the two-tier interface offers a portable, high-performance interface to the compiler over various network hardware.

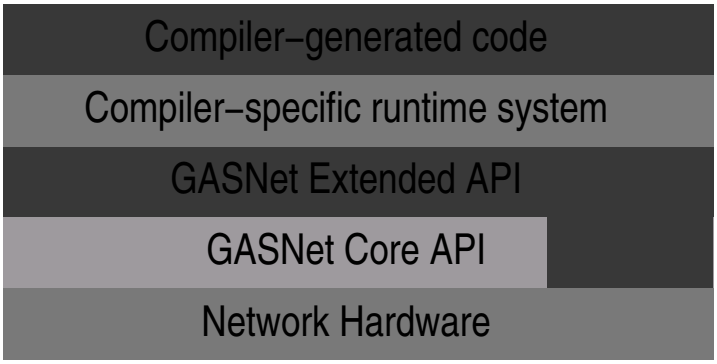

Figure 2: GASNet communications system layered approach: parts of the GASNet core may be bypassed to offer optimized hardware operations.

When GASNet is ported to a network such as Myrinet, the lowest-level communication layer available is generally used to implement the GASNet *conduit* for that network (GASNet offers conduit implementations for many high-performance networks – code is available online [2] and includes our implementation of the Firehose algorithm in the GM conduit). GM is a Myrinet low-level message-passing system that includes a driver, a control program for the Myrinet interface, a network mapper and the GM API (library and header files) [2]. GM provides protected user-level access to the NIC; reliable, ordered delivery of messages; recovery from transient network problems; scalability to thousands of nodes and a low CPU utilization on Myrinet hosts [18].

4.2. Testing Methodology

Empirical results were collected to compare the performance of the Firehose algorithm and two flavors of the Rendezvous algorithm: *with-unpin* and *no-unpin* of remote pages. Upon sending a put, the Rendezvous algorithm pins the source memory (if necessary) and sends an Active Message to the remote node indicating the destination memory to be pinned. Once the put is complete, the Rendezvous *with-unpin* strategy sends an additional Active Message to unpin the remote pages. Although the Rendezvous *no-unpin* approach is used by some Myrinet communication layers [4], it permits clients to request pinned memory until they've exhausted physical memory. Omitting the unpin step does not constitute a viable approach for GAS languages, but serves as a good metric when comparing to Firehose. In fact, we show that Firehose consistently outperforms Rendezvous, even without unpin, when the working set memory size is within Firehose's M parameter. It should also be noted that our Rendezvous implementation benefits from the highly tuned non-blocking primitives available in GASNet and is also augmented with caching of local pinned pages to avoid unpin-repin costs for local pages (a significant performance gain when issuing many successive Rendezvous puts from overlapping memory locations).

For the following tests, Firehose assumes M = 400MB of pinnable memory and MAXVICTIM = 50MB, which means each node owns $\lfloor \frac{400\text{MB}}{4096*(nodes-1)} \rfloor = \lfloor \frac{102400}{(nodes-1)} \rfloor$ firehoses to each remote node, and the limit on total pinned memory is M+MAXVICTIM = 450MB. The tests are run long enough for Firehose to reach a steady state. The bucket size is set to single-page buckets to provide an upper bound on the overhead in managing Firehose metadata.

4.3. Application Benchmarks

Firehose was benchmarked using two parallel applications implemented in Titanium: Cannon's Matrix Multiplication and a Bitonic Sort. In Cannon's algorithm [11], nodes are laid out on a $N \times N$ mesh and multiplication results are optimized to be accumulated locally. If results are accumulated on remote nodes, put operations are used (the results below use 4 nodes). The bitonic sort algorithm divides N integers equally among 8 nodes and runs a parallel quick sort. Results are then merged using a compare and exchange algorithm.

Application results are presented in Table 3. Because of the relatively small working set, Firehose never needed to detach firehoses or unpin pages in the firehose move operation, it merely attached new firehoses that were free. In the Firehose results shown, firehose moves constitute less than one percent of all puts, and vary in performance for both applications. Since Cannon makes heavier use of firehose moves (almost 6 times more than Bitonic sort),

[2] With the anticipated release of GM 2.0, GASNet/GM will also support one-sided gets through the Firehose algorithm.

Application	*Total Puts*	*Pinning Strategy*	*Total Runtime*	*Type of put*	*Number of puts*	*(%)*	*Avg. Put Latency*
Cannon Matrix Multiply	1 500 000	Rendezvous *with unpin*	5460.00s	pin-put-unpin	1 500 000	(100%)	5141 us
		Rendezvous *no-unpin*	797.45s	pin-put	1 500 000	(100%)	34 us
		Firehose	780.81s	move firehose	2 934	(0.2%)	46 us
				one-sided	1 497 066	(99.8%)	14 us
Bitonic Sort	2 125 000	Rendezvous *with unpin*	4740.05s	pin-put-unpin	2 125 000	(100%)	522 us
		Rendezvous *no-unpin*	289.45s	pin-put	2 125 000	(100%)	33 us
		Firehose	255.25s	move firehose	518	(0.02%)	54 us
				one-sided	2 124 482	(99.98%)	15 us

Table 3: Application performance with the Firehose and Rendezvous strategies

firehose moves are more likely to target a previously pinned memory location. This is shown in the difference in average firehose move times (46 us *vs* 54 us) – a move request satisfied by attaching firehoses to buckets in the FIFO queue is completed within 5us while an actual pin operation delays the request by 40 us. Results obtained show that reuse of buckets in the FIFO queue is beneficial as the average move time as part of the remote firehose handler for both applications is respectively 5us and 14us (significantly less than the average pin time of 40us). On the requesting node, the total overhead for building the firehose list, updating the firehose table and other bookkeeping tasks consumes less than a microsecond. The remaining time is shared between the messaging overhead for the Active Message and other GM/GASNet overheads. While the request/reply tends to be in the 20us range, this result varies with node attentiveness to the network.

Results show the average put latency under Firehose is twice as good as the Rendezvous *no-unpin* approach, and several orders of magnitude better than the Rendezvous *with-unpin* approach. Moreover, both applications show that the common case of using one-sided puts over firehose moves is achieved nearly 100% of the time, and both applications show a noticeable improvement in total running time when using Firehose (applications which are less latency tolerant or more bandwidth sensitive should show an even greater improvement).

4.4. Synthetic Microbenchmarks

We've developed a synthetic benchmark which measures the Firehose and Rendezvous approaches to compare the achieveable bandwidth and start-to-finish latency for single operations.

- **Small-message latency**. Using 8-byte puts between two nodes, the latency performance of Firehose under various working set memory sizes is shown by scattering puts in a uniform random pattern over increasing remote memory spaces. The put operations initiated on each node are sourced from a random memory location within an area of memory of size MAXVICTIM in order to better correlate increasing destination memory space on firehose movement. Figure 3 shows how Firehose consistently outperforms the best Rendezvous solution by 30 us when the remote access working set size does not exceed M+MAXVICTIM (*with-unpin* is not shown since its numbers are steadily in the 6ms range, which is over 100 times slower). The dramatic reduction in put latency is a direct consequence of Firehose's ability to remove extraneous messages from the critical path in the common case of a firehose hit, and use entirely one-sided DMA. Past the M+MAXVICTIM point (450 MB), the Firehose latency increases sharply, passing Rendezvous *no-unpin* once 3% of the 8-byte puts require unpinning, and approaching the Rendezvous *with-unpin* performance (not shown) – the high 6ms penalty for each unpin operation quickly dominates all other factors. It should be noted that for working sets larger than M+MAXVICTIM, the Rendezvous *no-unpin* algorithm is "cheating" in that it has more than M+MAXVICTIM pages (450MB) simultaneously pinned (because it never unpins any remote or local pages) and if M+MAXVICTIM constitutes a hard limit on pinnable physical pages, Rendezvous *no-unpin* would have crashed before reaching this point. Under implementations where the unpin operation is less expensive, we expect Firehose to display better performance characteristics past the M+MAXVICTIM point. Additionally, the hit rate and performance should be even better under a more realistic access pattern that exhibits better spatial and temporal locality within the working set than uniform random distribution.

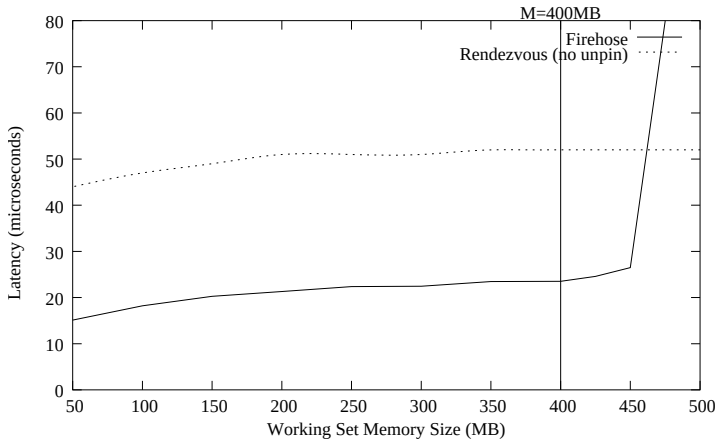

Figure 3: 8-byte put latency over increasing working set memory size (M = 400MB)

- **Large-message bandwidth**. Figure 4 shows results from a test where put operations of 64 KB are sent from one node to another, covering an increasing memory space in a uniform random distribution of targets. For working sets smaller than M, Firehose effectively provides a 100% hit rate and all puts proceed at full DMA bandwidth with no handshaking overheads – this corresponds to the case where the working set of remotely referenced pages fits within pinnable physical memory, and we expect this to be the common case for real scientific applications (note the actual value of M is a tunable Fire-

hose parameter whose optimal value is site-dependent). When the working set exceeds M, the Firehose hit rate decreases (increasing the amount of handshaking required to handle misses) and performance degrades gracefully, eventually approaching the performance of Rendezvous *with-unpin* at very large working set sizes (at which point the application is probably swapping to disk to accomodate the working set which exceeds physical memory). Unpin operations are relatively less expensive in the large message test since the latency to complete a one-sided 64 KB put is much higher in comparison to an 8 byte put. Rendezvous *no-unpin* is shown for comparison purposes, however it should again be noted that beyond M+MAXVICTIM (450MB) the Rendezvous *no-unpin* algorithm has exceeded the limit on pinnable physical pages (and therefore is not really a viable solution at that point).

- **Peak bandwidth**. Peak bandwidth for a range of put sizes is reported in Figure 5 for both algorithms. This test uses repeated puts to/from the same location, so Firehose effectively provides a 100% one-sided hit rate. Firehose beats Rendezvous *no-unpin* by a noticeable margin because it requires no handshaking messages before performing a DMA. Rendezvous *with-unpin* consistently performs dismally due to the high cost of unpinning on Myrinet which must be paid on every operation.

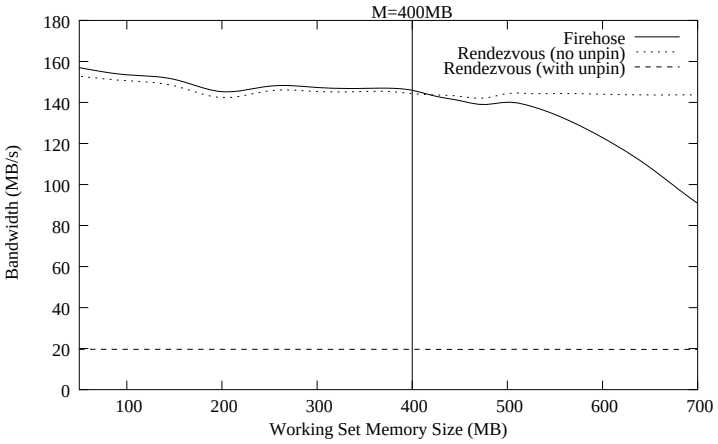

Figure 4: 64KB put bandwidth over increasing working set memory size (M = 400MB)

5. Related Work

User-level networking has provided many benefits by removing the operating system and intermediate buffering from the critical path. Its viability has been demonstrated through the implementation of user-level networking libraries on commodity networks, such as U-Net[26], FM[22], BIP[23] and fbufs[12]. In some cases, a suggested coupling of the network interface with the processor's TLB (as in Stanford FLASH[16]), or the integration of a TLB directly on the interface [28] have proven useful in maximizing remotely addressible user memory. These approaches are important in that they do not require the programmer to be directly involved in managing DMA-enabled user memory. SHRIMP's Virtual Memory-Mapped Communication system [8] allowed users to

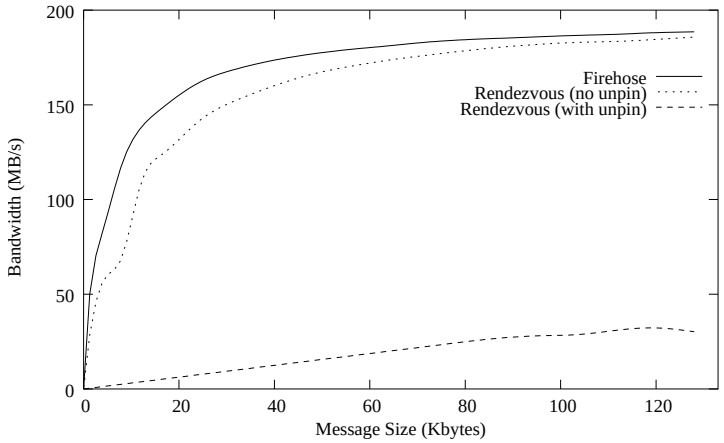

Figure 5: Peak Bandwith for Rendezvous and Firehose

export pinned pages to remote nodes which then mapped them into the page table, and provided hardware support for translating CPU store operations into RDMA puts. It is widely accepted that registration costs on pinning-based networks constitute an important bottleneck. As a part of the PM communications library, registration is addressed using a *Pin-down Cache*[25], where mappings of locally pinned pages are cached on a LRU basis. Although this constitutes an improvement over blindly pinning and unpinning for every remote memory operation, each of these operations require a previous network roundtrip such that both source and destinations are pinned.

6. Conclusion

The Firehose algorithm presented in this paper is motivated by the inadequacy of current pinning-based networks to support remote memory operations which access large portions of globally-shared memory. Efficient implementation of GAS languages requires communication systems capable of providing low-overhead and low-latency for small, non-blocking remote puts/gets and high-bandwidth, zero-copy transfers for large puts/gets. On networks such as Myrinet/GM, these goals are best achieved through the use of fully one-sided RDMA operations, but the need to perform explicit DMA registration on the remote host (without compromising the host's physical memory resources by simply pinning everything at startup, or forfeiting the performance benefits of truly one-sided communication by imposing handshaking and synchronization) poses an interesting algorithmic challenge.

The Firehose algorithm presented successfully exposes one-sided, zero-copy communication as a common case, while minimizing the number of host-level synchronizations required to support remote memory operations, and amortizes the cost of synchronization and pinning over multiple remote memory operations. Firehose reaps the performance benefits of a Pin-Everything approach in the common case (without the drawbacks) and reverts to Rendezvous-like behavior to handle the uncommon case. Empirical results with synthetic and application benchmarks demonstrate that the cost of handshaking and memory registration in Firehose is negligible when the set of remotely referenced memory pages on a given node is smaller than the physical memory, and in applications with larger working sets the performance degrades gracefully

and consistently outperforms conventional approaches. We believe the Firehose algorithm is a near-ideal solution to the DMA registration problem presented by pinning-based networks such as Myrinet for GAS language implementations, and furthermore Firehose is sufficiently general that it should prove to be a useful strategy for implementing other communication systems with related goals on similar networks.

7. Future Work

Future work includes evaluating the performance of Firehose with get operations, once Myrinet releases the eagerly anticipated version of GM which will support one-sided DMA gets. Everything in the Firehose algorithm and implementation generalizes naturally to gets, once the underlying transport provides that capability (in the meantime, gets are implemented using an Active Message which initiates a Firehose put back to the requestor).

We are also interested in implementing the Firehose algorithm on other pinning-based high-performance networks with DMA capabilities – Infiniband [3] is one such potential target.

8. Acknowledgements

This work was supported in part by the Department of Energy under DE-FC03-01ER25509 and DE-AC03-76SF00098, by the National Science Foundation under ACI-9619020 and EIA-9802069, and by the Department of Defense. The information presented here does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred. We'd also like to thank Boon Thau Loo for providing the Titanium application code, and Kathy Yelick, Jarek Nieplocha, Paul Hargrove, and Myricom for their support and constructive suggestions.

References

[1] Berkeley UPC project home page. http://upc.lbl.gov.

[2] GASNet home page. http://www.cs.berkeley.edu/~bonachea/gasnet.

[3] Infiniband trade association home page. http://www.infinibandta.org.

[4] MPICH-GM implementation, v1.2.1..7b. http://www.myrinet.com.

[5] Titanium home page. http://titanium.cs.berkeley.edu.

[6] MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[7] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. In *IPDPS 2003*, 2003. http://upc.lbl.gov.

[8] M. A. Blumrich, R. D. Albert, Y. Chen, D. W. Clark, S. N. Damianakis, C. Dubnicki, E. W. Felten, L. Iftode, K. Li, M. Martonosi, and R. A. Shillner. Design choices in the SHRIMP system: An empirical study. In *Proc. of the 25th Annual Int'l Symp. on Computer Architecture (ISCA'98)*, 1998.

[9] 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.

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

[11] L. E. Cannon. A cellular computer to implement the kalman filter algorithm, 1969.

[12] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Symposium on Operating Systems Principles*, pages 189–202, 1993.

[13] J. Duell. Memory management in the UPC runtime, 2002. http://upc.lbl.gov.

[14] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC specification, v1.0, February 2001. http://upc.gwu.edu.

[15] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Tech Report UCB/CSD-01-1163, U.C. Berkeley, November 2001.

[16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, 1994.

[17] A. Mainwaring and D. Culler. Active message applications programming interface and communication subsystem organization. Tech Report UCB/CSD-96-918, U.C. Berkeley, 1995.

[18] Myricom. *The GM Message Passing System*. Myricom, Inc, GM v1.5 edition, July 2002.

[19] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Proc. RTSPP IPPS/SDP'99*, 1999.

[20] 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.

[21] R. Numrich and J. Reid. Co-array fortran for parallel programming. In *ACM Fortran Forum 17, 2, 1-31.*, 1998.

[22] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. 1995.

[23] L. Prylli and B. Tourancheau. Bip messages user manual, 1997.

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

[25] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *12th Int. Parallel Processing Symposium*, pages 308–315, 1998.

[26] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *15th ACM Symposium on Operating Systems Principles*, pages 40–53, 1995.

[27] 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.

[28] M. Welsh, A. Basu, and T. von Eicken. Incorporating memory management into user-level network interfaces. Technical Report TR97-1620, 13, 1997.

[29] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.