# Scalable Memory Registration for High Performance Networks Using Helper Threads

Dong Li  Kirk W. Cameron
Department of Computer Science, Virginia Tech, USA
{lid, cameron}@cs.vt.edu

Dimitrios S. Nikolopoulos
FORTH-ICS and University of Crete, GREECE
{dsn}@ics.forth.gr

Bronis R. de Supinski  Martin Schulz
Lawrence Livermore National Lab, USA
{bronis, schulzm}@llnl.gov

## Abstract

Remote DMA (RDMA) enables high performance networks to reduce data copying between an application and the operating system (OS). However RDMA operations in some high performance networks require communication memory explicitly registered with the network adapter and pinned by the OS. Memory registration and pinning limits the flexibility of the memory system and reduces the amount of memory that user processes can allocate. These issues become more significant on multicore platforms, since registered memory demand grows linearly with the number of processor cores. In this paper we propose a new memory registration/deregistration strategy to reduce registered memory on multicore architectures for HPC applications. We hide the cost of dynamic memory management by offloading all dynamic memory registration and deregistration requests to a dedicated memory management helper thread. We investigate design policies and performance implications of the helper thread approach. We evaluate our framework with the NAS parallel benchmarks, for which our registration scheme significantly reduces the registered memory (23.62% on average and up to 49.39%) and avoids memory registration/deregistration costs for reused communication memory. We show that our system enables the execution of problem sizes that could not complete under existing memory registration strategies.

## Categories and Subject Descriptors

C.3.2 [*Computer-Communication Networks*]: Network Operations—Network Management; D.4.1 [*Operating Systems*]: Process Management—Threads; D.4.4 [*Operating Systems*]: Communication Management—Network Communication

## General Terms

Design, Management, Performance

## 1. Introduction

Continuing progress in hardware technology enables the integration of multiple processor cores on a single chip. Chip multiprocessors (CMPs) bring opportunities and challenges to high performance applications, programming models, and operating systems (OSs). CMPs allow aggregation of multiple tasks within a node in order to save system power and even to improve performance [20]. CMPs also enable novel techniques that leverage idle cores to boost performance [6, 32]. However, CMPs increase pressure on shared resources at the node level, such as the network interface and the memory hierarchy. Several software [29] and hardware [31] schemes attempt to address this problem to achieve scaling to a steadily increasing number of cores.

Many high performance networks, such as Infiniband [13] and Myrinet [4], leverage remote direct memory access (RDMA) to achieve high bandwidth transfers. RDMA enables data transfer from the address space of an application process to a peer process across the network fabric without requiring host CPU involvement. With most interconnects, RDMA operations require *memory registration*, which explicitly identifies the memory used in these transfers. Further, every page of the communication buffer must be pinned to prevent swapping by the OS. Pinning memory limits adaptability of the memory system and reduces the amount of memory that user processes can allocate. Unnecessarily pinning pages can underutilize memory. Pinned memory is also a precious resource: some systems constrain the maximum amount of memory that a user process can lock and some network adapters, such as the Infiniband Host Channel Adapter (HCA), limit the number of simultaneously pinned separate regions of contiguous virtual pages. The increasing popularity of GPUs for HPC makes the memory pinning problem even worse, since GPUs may also require pinning host memory mapped into the GPU address space to perform memory copies concurrently with kernel execution.

CMPs further increase the challenge of the memory registration problem since the demand for registered memory grows linearly with the number of processes on the node. Some resource managers like SLURM [18] and Torque [1] can limit the amount of memory that jobs can pin. However, an HPC application can require a large amount of pinned memory (see Section 6). If we attempt to use more than one task per node on fewer nodes, aiming to reduce power consumption or improve performance through shared memory communication [20], the application can easily exceed the pinned memory limit, which prohibits using the idle cores.

The naive option would be to deregister the communication buffer after the RDMA operation completes. However, HPC applications often exhibit repeating communication patterns [8, 14]. Also, memory registration/deregistration is expensive [21, 24] since it requires a system call and an overall linear cost in the number of pages [33]. Thus, this simple solution can incur a significant performance loss by failing to reuse the buffer.

We propose a novel scalable memory registration strategy that is sufficiently general to satisfy the most demanding RDMA requirements. Our approach hides the cost of dynamic memory manage-

*CF'11*, May 3–5, 2011, Ischia, Italy.
Copyright © 2011 ACM 978-1-4503-0698-0/11/05...$10.00

ment by offloading all dynamic memory registration/deregistration requests to a dedicated memory management helper thread, while sustaining performance. Our solution makes more memory available to the application, which will be critical as systems come to have less memory per core. Dynamic memory management via the helper thread introduces several challenges. First, we must identify and extract parallelism that exists between the main application thread and the helper thread in order to hide operation latency through proactive memory registration and asynchronous deregistration. Thus, the helper thread must predict *when* it should register *which* memory locations. Second, we must minimize communication and synchronization between the main thread and the helper thread to avoid increasing latencies on the communication critical path. Third, a new memory registration scheme should be transparent to the application and the programming API (e.g., MPI). A fully transparent and dynamic solution should not require profiling information, compiler support, programmer hints, source code modification or hardware support. The helper thread mechanism should apply to any memory registration/deregistration implementation and RDMA communication protocol.

This paper makes the following contributions:

- A novel helper thread strategy to reduce the size of registered memory in high performance networks based on memory pinning;
- A novel context-aware mechanism to predict when to register specific memory regions for the purposes of communication;
- Time-oriented techniques that avoid increasing critical path latencies when using a communication helper thread;
- An investigation of helper thread design policies that efficiently leverage idle cores;
- Performance analysis of different memory registration schemes and a detailed study of registered memory usage of large-scale parallel applications under strong and weak scaling, including aggregation of multiple tasks per node.

Our communication context-aware predictor achieves high accuracy with a short learning process. We apply our helper thread strategy and predictor to several NAS benchmarks, including communication intensive applications. Our mechanisms reduce the registered memory size by up to 49.39% and by 23.62% on average, while avoiding memory registration/deregistration overhead for reused communication memory.

## 2. Background

High performance networks combine DMA engines in the network adapter with user level communication protocols to achieve low CPU utilization, high bandwidth, and low latency. In all widely used interconnection technologies, the DMA engine transfers data from physical addresses in main memory to the network adapter. Virtual addresses of communication buffers must be translated into physical addresses. The memory pages corresponding to these addresses must be pinned for the duration of the transfer to prevent the OS from paging them to disk. *Memory registration* refers to the process of address translation and pinning. We now outline common memory registration strategies.

**Bounce Buffer:** MPICH [22], MVAPICH [12] and Open MPI [11] use the *bounce buffer* method in which the connection setup phase pre-registers a set of RDMA buffers and exchanges their addresses and associated keys between communication peers. MPI communication then copies from/to the user's buffer in user space to/from the bounce buffers and uses RDMA on the bounce buffers to transfer the data. This method limits the registered memory size to the size of bounce buffers. It incurs registration costs only at startup, which potentially amortizes them over many RDMA operations. However, this method incurs significant memory copy costs, particularly for large messages. Thus, it best suits small messages,

```
/* Sender: */

/* The sender and the receiver begin
  communication at the same time */
MPI_Barrier(...);

for(iter=0; iter<10; iter++)
{
 MPI_Send(..., buffer_1, ...);
 random_comp(); /* lasting 1 sec */
 MPI_Send(..., buffer_2, ...);
 random_comp();
 MPI_Send(..., buffer_3, ...);
 random_comp();
}
```

```
/* Receiver: */

/* The sender and the receiver begin
  communication at the same time */
MPI_Barrier(...);

for(iter=0; iter<10; iter++)
{
 MPI_Recv(..., buffer_1, ...);
 random_comp(); /* lasting 1 sec */
 MPI_Recv(..., buffer_2, ...);
 random_comp();
 MPI_Recv(..., buffer_3, ...);
 random_comp();
}
```

Figure 1: Microbenchmark pseudocode

including messages for protocol control. We could apply our helper thread strategy to small messages to eliminate the bounce buffers. However searching preregistered buffers and identifying the communication call site, as with our helper threads (see section 3), entails high enough overhead for small messages that the bounce buffers are preferable. Thus, we only use helper threads for messages larger than a threshold (16KB in our experiments).

**Registering Memory on Demand:** Many RDMA libraries register memory on demand [11, 12, 22]. This alternative scheme registers user buffers at the communication call site, which adds the overhead of memory registration to the communication critical path. Some implementations [11, 22] leave user buffers pinned after the communication, which can amortize the cost if the user buffer is reused often. However, this choice increases registered memory size, even if the buffer is only used once.

**Hardware-Assisted Memory Registration:** Quadrics [25] combines a hardware TLB in the network adapter with a tuned virtual memory subsystem to allow hardware to pin pages, to initiate page faults and to track page table changes. This approach avoids memory registration overhead and minimizes pinned memory size. However, it increases hardware complexity and cost and requires platform and OS-specific modifications.

Figure 1 depicts a microbenchmark that captures the limitations of prior memory registration schemes that our helper thread approach overcomes. In this common execution pattern of HPC applications, point-to-point communication repeatedly exchanges data between locations. Figure 2 shows the registered memory size over time on the top and the communication time on the bottom when we run the microbenchmark on two nodes connected with Infiniband using Open MPI 1.4 and two memory registration schemes (section 6 describes additional hardware configuration details). We measure sender side time for a 5MB message. The bounce buffer occupies region 1 of registered memory, while memory registered for communication with large messages occupies region 2. With the "leave pinned" registration on demand approach, the registered memory increases steadily and later remains constant. The "no leave pinned" approach registers less memory, but takes 25.83% longer. Our helper thread approach breaks this dichotomy.

As further motivation, one of our experimental platforms has 8 cores and 8GB of physical memory per node. If we limit the registered memory size per process to 3GB, using just two MPI processes per node consumes 75% of the available physical memory, leaving very little physical memory available to run realistic applications and services. On a system with paging, physical memory limitations would entail a large performance loss, whereas on a system without paging (e.g., uClinux), applications would be limited to using 2 GB of physical memory, which amounts to 256 MB per core. Furthermore, even with a large memory registration limit of 3 GB for communication purposes, we cannot meet the memory demand of some communication-intensive applications. For example, we cannot run problem size D of NAS PB FT with a 2×16 2D processor layout and the standard Open MPI installation. As Figure 3 shows, a single FT task requires 3.58GB of pinned memory.

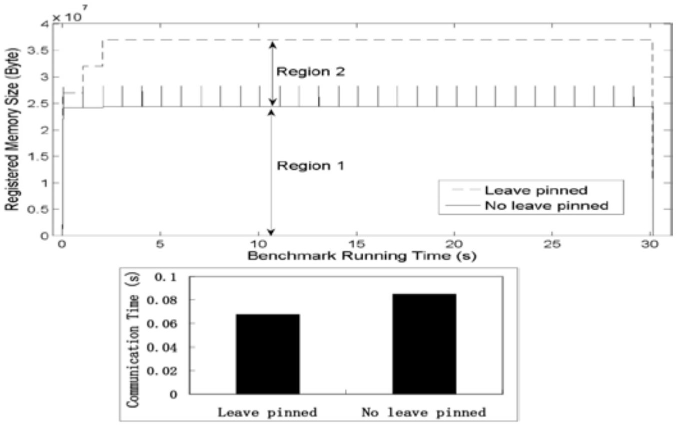


Figure 2: Traditional memory registration sizes and communication times

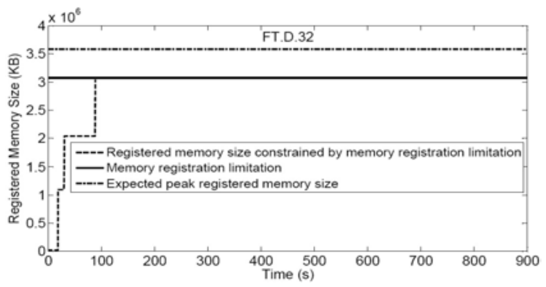


Figure 3: Memory registration limit prevents application execution

As we show in section 6, our approach enables the execution of this application even with the memory registration limit.

## 3. Design

HPC applications interleave communication and computation and attempt to overlap communication with computation using techniques such as non-blocking send-receive operations and progress threads [29]. Our helper thread scheme offloads memory registration overhead to overlap it with computation. Thus, we register and deregister memory outside of communication phases. We adopt a prediction-based approach. Our design attempts to predict execution points that require memory based on iterative execution patterns in HPC applications [8].

### 3.1 Context-Aware Predictor

We present a time-based, context-aware predictor that attempts to identify iterative usage patterns of each specific registered memory region from MPI message streams and to predict the future registration time of each region. The predictor must dynamically detect the registration period of each region, which we then use to predict the next registration time.

Alternatively, we could leverage temporal patterns in MPI messages [8], thus saving registered memory without considering the time between communication operations. Specifically, we could register a set of recently referenced memory regions, similarly to an LRU cache. However, this method cannot minimize registered memory size. An application may have large time gaps between RDMA communication operations, during which we can deregister previously registered memory. We require the time information to design an effective strategy to minimize registered memory size throughout the lifetime of an application.

A simple predictor design would record the registration time for each memory region and then use the time between consecutive

```
main computation loop
{
  ...
 inner loop1
 {
  ...
  MPI_Comm1 ( ... , buf1 , ... ) ;
  ...
 }
  ...
 inner loop2
 {
  ...
  MPI_Comm2 ( ... , buf2 , ... ) ;
  ...
 }
}
```

Figure 4: Loop nests

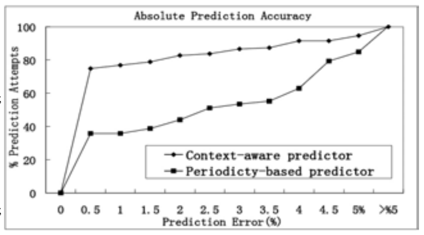


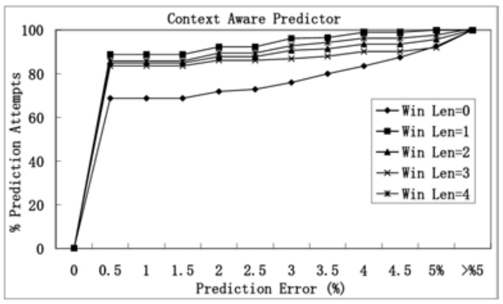
Figure 5: Prediction accuracy comparison

registrations. However, this design cannot handle complex memory access patterns. Different control paths may use the same memory region with different periods. Thus, we associate call site information with each registered memory region to detect the periodicity, instead of only identifying the memory region. We hash the call site stack frame addresses and the registered memory region address in a call site's *signature*. This solution distinguishes between uses of the same memory region with different registration periods. To minimize the impact on the application, the hashing operations are performed by the helper thread leaving only the recording of stack frame addresses in the critical path.

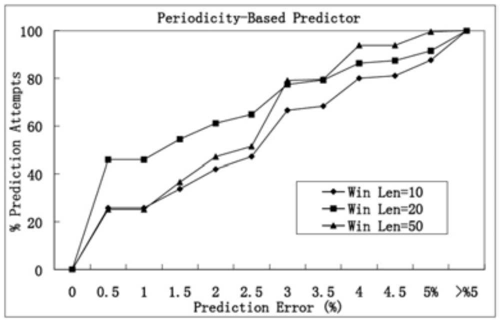
Using call site addresses may poorly predict the registration period for cases such as the nested loops shown in Figure 4. We cannot predict the registration time of the communication call site of the first iteration of inner loop 1 from the per iteration time of that loop. However, that time is the registration period of all other instances of that call site. The first iteration of inner loop 1 follows a full execution of loop 2 except for when it is part of the first iteration of the outer loop. This call site has three different registration periods. Thus, we extend the communication context information to include the previous communication point to handle call sites with multiple periods. By hashing call site information from the previous (i.e., communication type and memory address) and current (i.e., stack frame addresses and the registered memory address) communication call sites in the signature, we distinguish between communications with the same stack frame addresses but different registration periods due to control flow differences.

We use a communication context window to record communication streams (Section 4.2). We analyze the sensitivity to the window length (i.e., how many adjacent call sites we consider) for the NAS CG benchmark, as Figure 6(a) shows. "Win Len=0" indicates that we only use current call site information, which decreases the percentage of predictions with error less than 0.5% by 20%. Thus, we must consider previous call sites although accuracy increases little if we use more than the immediately preceding one. Based on this and similar results for five other NPB benchmarks, we use a window length of two, which incurs little additional cost for potentially greater accuracy in rare cases, for the rest of the paper.

We compare our predictor with a periodicity-based predictor that records a window of communication streams to detect iterative patterns [8]. It compares a communication subsequence with another subsequence shifted by $m$ samples, where $m$ is a parameter that is less than the window size. It detects an iterative pattern with period $m$ if the two subsequences are identical. We apply this technique to detect patterns and their time periods to predict memory registration time. We analyzed the sensitivity of this predictor to its window length. We found that it is highly sensitive to this parameter at any desired accuracy, as Figure 6(b) shows. Further, it has a long learning process that requires a full window and a complete pattern. Our predictor has a short learning process since its context window is short and its predictions are independent of the pattern frequency.



(a) Context-aware predictor



(b) Periodicity-based predictor

Figure 6: Sensitivity analysis of the two predictors

Figure 5 compares results of our predictor with the periodicity-based one for six NAS parallel benchmarks. Both approaches are accurate: 84.93% of periodicity-based and 94.68% of our predictions are within 5% of the observed period. However, 74.89% of our predictions have error less than 0.5% while only less than 40% of the periodicity-based predictions achieve error that low.

**Discussion:** Our approach so far assumes that the workload per iteration is stable and, therefore, memory registration is periodic. Some HPC applications (e.g., AMG in the Sequoia benchmark suite [17]) do not have a constant workload across iterations. Although these applications do exhibit iterative memory registration patterns, the registrations may not be periodic. In this case, we use the shortest time between registrations in the same context.

## 3.2 Helper Thread Design

**General Description:** Given the predicted registration time, we can design an algorithm for the helper thread. This design must balance between registering a memory region before a communication operation uses it and delaying memory registration as long as possible. If we do not register memory before a communication uses it, the main thread must register it on demand at the cost of additional latency; registering memory too soon increases the peak memory registration size.

Our design uses two queues: a deregistration queue (DQ) and a registration queue (RQ). The helper thread processes these queues in turn and switches between them based on a cost estimate for processing items from each. We use fine-grained time control to ensure that we achieve the necessary balance.

**Memory Deregistration:** The main thread places information about registered memory (registration time, address, size and adjacent communication point information) into the DQ, after the completion of a communication operation. The helper thread removes this information and computes its signature, which it uses to extract information from the *call site hash table*. The call site hash table stores predicted registration periods for call sites. The helper thread must complete the registration of each call site before its *registration deadline*, which is its last registration time plus its period.

The helper thread uses the registration deadline to determine if it should deregister a memory region, which multiple communication call sites may use. We deregister a memory region and insert it into the RQ with its registration time set to the registration deadline if a *deregistration condition* is satisfied. This condition requires that the current time plus the estimated processing delay for memory registration is not later than the earliest registration deadline associated with the region. The helper thread finds the nearest future registration time for a region in the RQ after registering it and updates the earliest registration time. The first time we register a memory region, we do not have a predicted registration period for it. Thus, we always deregister it, which saves registered memory from those operations that register regions only once. After processing one item from the DQ, the helper thread switches to the RQ to check registration requests, in case any registration deadline is near.

**Memory Registration:** the RQ is ordered based on the registration time of each item in the list. The helper thread processes the RQ starting from the item with the earliest registration time. It delays registering a memory region and switches to DQ processing if the following *registration condition* is satisfied: the current time plus the estimated time for processing one DQ item plus the estimated processing delay for memory registration is earlier than the region's registration deadline. Otherwise, the helper thread registers the memory and updates its associated earliest registration time. We track registered memory information in a *registration cache* for fast reference.

**Adjusting Registration Deadlines:** Some HPC applications have communication-intensive phases in which multiple communication sites have similar registration deadlines. Since the helper thread tries to delay memory registration until the deadline, it may not be able to register all memory regions on time, in which case we incur additional latency on the communication critical path. We avoid this cost by adjusting deadlines when we insert a new item into the RQ. The registration time gap between any two items must be longer than the cost of processing one RQ item plus the cost of processing one DQ item, so that we can process both the DQ and the RQ in time to make the registration deadlines. We decrease registration deadlines when necessary to meet this condition.

**Time Control and Estimation:** Our design critically depends on estimated DQ and RQ processing times. The following inequality describes our registration and deregistration conditions:

$$t_c + t_p \leq t_d \tag{1}$$

Where $t_c$ is the current time, $t_p$ is the processing time by the helper thread and $t_d$ is the registration deadline.

We can perform deregistration if the inequality is true. For the deregistration condition, $t_p$ includes: (1) the deregistration cost ($t_{deregister}$); (2) the cost of inserting the registration request into the RQ ($t_{insert}$); (3) the registration cost ($t_{register}$); and (4) the cost of updating the earliest registration time for a memory region ($t_{update}$). Similarly, we can delay registration and switch to processing the DQ if the inequality is true. For the registration condition, $t_p$ includes: (1) the cost of searching the hash table and updating call site information ($t_{hash}$); (2) the cost of searching the registration cache ($t_{cache}$); (3) $t_{deregister}$; (4) $t_{insert}$; (5) $t_{register}$; and (6) $t_{update}$.

We model the memory (de)registration cost with $t_{(de)register} = a * p + b$, where $a$ is the (de)registration cost per page, $b$ is the overhead per operation, and $p$ is the size of the memory region in pages [24]. We determine $a$ and $b$ through linear regression. The RDMA library or device driver may include other optimizations to improve the performance of (de)registration. While these optimizations can make the cost of (de)registration irregular, we attempt to

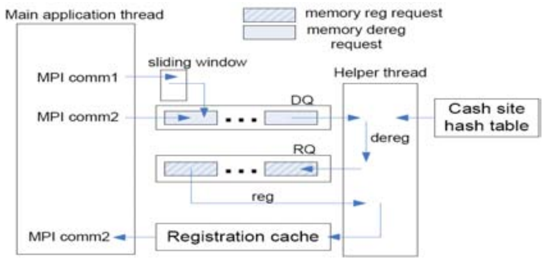


Figure 7: An illustration of the helper thread implementation

keep our algorithm independent of these optimizations by estimating the maximum potential cost. Alternatively, we could use a more complex (de)registration model.

The number of queue items that we must compare in each step impacts $t_{insert}$ and $t_{update}$, which represent the time for queue operations. Each comparison has a nearly fixed number of instructions, which is easy to estimate. We conservatively assume that we compare every item in the queue for the insert/update operation to simplify the overall estimates, although an algorithm like binary insertion may involve fewer items. We can also easily estimate $t_{hash}$ if we assume that hash collisions are rare. We estimate $t_{cache}$, which is related to the number of cache items that the algorithm must compare, similarly to $t_{insert}$ and $t_{update}$.

## 4. Implementation

Figure 7 illustrates our helper thread implementation, which we discuss in this section.

### 4.1 Synchronization Issues

Thread synchronization latency is an important design consideration for helper thread software. We implement the DQ as a shared, lock-less circular queue. The main thread moves the head pointer when inserting an item and the helper thread moves the tail pointer when removing an item. Initially, the head pointer and the tail pointer point to the same position. The helper thread compares the two pointers to determine if the DQ has new items. If so, it moves the tail pointer forward and dequeues one item for processing.

A circular queue needs a lock to prevent the head pointer from overwriting unused data. However, most HPC applications interleave communication with computation. Since the computation phase is usually much longer than the processing of a queue item in the helper thread, we usually move the tail much faster than the head so overwriting is unlikely. Further, if an application has short (or no) computation phases, so that we move the head pointer more quickly then the time between the communication call sites is too small to perform memory deregistration and registration without performance loss. Thus, we can safely overwrite and, as a result, skip old queue entries. Currently, we only use one helper thread and hence we do not need a lock for the tail pointer. We also do not need a lock for the head pointer if the MPI implementation does not provide `MPI_THREAD_MULTIPLE` support, which ensures only one thread will access it at any time. Since the helper thread reads the head pointer, read after write (RAW) and write after read (WAR) hazards may happen. In our case, a RAW hazard only temporarily delays the processing of a new item, which does not add any latency to the critical path. A WAR hazard also has no negative impact since the helper thread tests the difference between the head and the tail pointers.

The main thread and the helper thread share the registration cache as well as the DQ. The helper thread puts registered memory information (address, size and a pointer to the registered memory region) into the cache prior to communication. The main thread obtains the registered memory region pointer from the cache. Thus, we must lock the memory region pointer. We use atomic instructions (compare and swap) to implement a lightweight lock and spin if it is held. With this lock, the synchronization latency is $0.072\mu s$ when two threads contend for the lock and $0.018\mu s$ without contention. This lock overhead is low since MPI communication latency for large messages (larger than our 16KB threshold) is at least tens of microseconds with state-of-art hardware [16, 30].

### 4.2 Predictor Implementation

The predictor must collect context information by using a sliding window with length equal to the number of communication call sites for which it constructs signatures. Each call site adds information to the window. When the predictor computes a call site signature, it obtains data for adjacent call sites from the window. We associate each call site leading to memory registration with a call site hash table entry. The hash table entry records the last memory registration time and the predicted period. We update the last memory registration time when we begin memory registration for the call site. We also compute the time between the current memory registration time and the last one, which we record as the new prediction if it is less than the current predicted period. We use the shortest observed time because effects such as networking perturbation, OS noise and computation workload vibration may make the period irregular. Choosing the shortest period reduces the probability that we miss the real registration deadline.

### 4.3 Overhead Analysis

We introduce four operations on the critical path of the main thread. First, we read the time stamp counter at the RDMA communication call sites, which typically costs between 150 and 200 clock cycles — or less than $1\mu s$ on a 1 GHz processor. Second, we acquire a synchronization lock for a registered memory pointer, which takes less than $0.1\mu s$, as discussed in Section 4.1. Third, we record the communication type and the buffer address in the context window, which is negligible compared to the cost of the actual MPI implementation and communication operation. Fourth, we record the stack frame address at the RDMA communication call site, which involves a few accesses of the frame pointer register and copying several bytes of data. This overhead is also small compared to the communication time of large messages. Nonetheless, we can overlap these overheads, other than collecting the context window information, with computation of the main thread for nonblocking MPI communication.

We measure our scheme's overhead with the microbenchmark shown in Figure 1. We use a 17KB message size (marginally larger than the minimum size at which we use our helper thread). We disable the helper thread, but keep the above four operations in the main thread, so that any performance benefit of the helper thread does not offset its overhead. We compare performance with the original implementation without the four operations. We run the test 100 times and compute the average communication time at the sender site and find that the overhead is 14.02%, which primarily comes from recording stack frame addresses. This test gives an upper bound on overhead since real applications are likely to be less communication-intensive or to use larger message sizes. The same microbenchmark finds much lower overhead with larger messages (4.58% for 1MB messages and 1.22% for 4MB messages).

## 5. Leveraging Idle Cores Efficiently

So far, we have assumed that each MPI task uses one helper thread for memory registration. In principle, helper threads use idle core cycles that are not used for computation or communication. We do not want to sacrifice application performance by dedicating cores to

|  | No helper thread | Conservative policy | Aggressive policy |
|---|---|---|---|
| SP.D.36 | 18.60 | 18.60 | 18.62 |
| CG.D.32 | 17.08 | 17.09 | 17.09 |

Table 1: Execution time (s) with different distribution policies.

helper threads. If no idle core is available, we do not use the helper thread. However, with an increasing number of cores per node in high-end computing environments, idle cores are often available due to application scalability limits [20]. In this section, we explore design alternatives for using idle cores to run helper threads.

When we have at least as many idle cores as helper threads, each helper thread solely occupies a core. This unshared core distribution allows each helper thread to explore registered memory reduction opportunities fully. We show in Section 6.3 how helper threads reduce the registered memory size for each MPI task under this distribution.

When we have fewer idle cores than helper threads, distributing helper threads across the idle cores becomes more challenging; we explore two possible policies. The *aggressive policy* tolerates the excess helper threads. Each task spawns a helper thread, without considering resource consumption. Helper threads that reside on the same core take turns conducting memory operations, depending on the OS scheduler. The *conservative policy* restricts helper threads to at most one per idle core through the coordinated elimination of helper threads in some processes if necessary.

The aggressive policy must adjust the registration/deregistration condition (Section 3.2) to consider concurrent helper threads. Assuming a round robin scheduling policy with a time slice of length $t_{slice}$ for helper threads on the same core, we change the registration/deregistration condition as follows:

$$t_c + t_p + n(t_{slice} + t_{cs}) \leq t_d \tag{2}$$

where $n$ is the number of helper threads on the core and $t_{cs}$ is the context switch overhead. We assume that an active helper thread does not know when its current time slice will end. We guarantee that we do not miss deadlines by requiring $t_d$ to be no earlier than the thread's *next* time slice to meet the (de)registration condition. This conservative assumption ensures that we can safely deregister any item for which the inequality holds now and register it again in a future time slice (the deregistration condition), or delay its registration to a future time slice (the registration condition).

The conservative policy uses our original registration/deregistration condition since it still has only one helper thread per idle core. Instead, we must coordinate tasks to decide which ones retain helper threads. We chose to provide helper threads for the tasks that register the most memory. We implement task coordination in shared memory, where each helper thread posts the registered memory size after finishing a fixed number of communication operations. We continue running the threads associated with the largest memory registrations and stop all others (we randomly break ties).

We apply both polices to NPB benchmarks SP and CG; Figure 8 and Table 1 show the results. Our test platform has 8 cores per node (see Section 6 for more details). We use 6 nodes with 6 tasks per node for SP. For 32 CG tasks, we use 6 nodes total: 2 with 6 tasks and 4 with 5 tasks (the task distribution is unbalanced because CG requires a power of 2 task count and we require that each node has at least 5 tasks so that we have fewer idle cores than tasks). Figure 8 shows results from the nodes with 6 tasks per node. We display only two time steps or two computation iterations in order to illustrate the variance clearly. We report the registered memory size for the entire node. Under the aggressive policy, we distribute the helper threads as evenly as possible between the available idle cores so each idle core hosts 3 helper threads.

The results show that the aggressive policy saves more registered memory on average than the conservative policy for SP. For

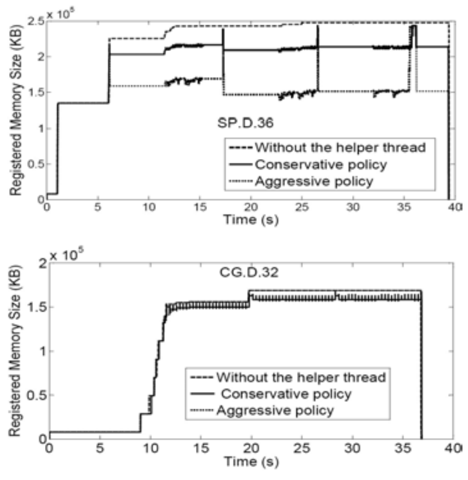


Figure 8: Registered memory sizes with different distribution policies

CG, the aggressive policy does not save memory, while the conservative policy does. SP does not generally interleave communication with computation and has a relatively long time between communication calls. Thus, the aggressive policy does not miss memory saving opportunities even though the helper threads share cores. The conservative policy saves less memory since only a third of the tasks have helper threads. Alternatively, CG is communication intensive so the aggressive policy cannot deregister memory without introducing latency into the communication critical path.

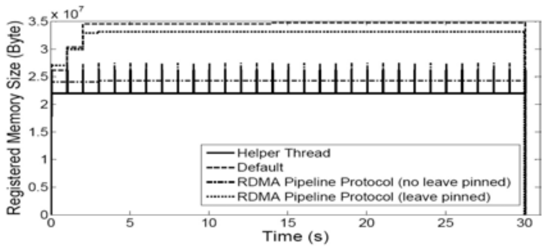
In general, performance of the distribution policies depends on application characteristics and OS scheduling. The aggressive policy assumes round robin scheduling with identical time slices. Linux determines the time slice length based on process priority. Although helper threads may have the same static priority, the dynamic priority may change, depending on how often the threads submit and wait on I/O requests. Thus, helper threads may not be scheduled round robin since the OS attempts to schedule threads with higher priority earlier. Our design also critically depends on time information. Unfortunately, a helper thread may be descheduled immediately after reading the clock, making the time stale when it resumes execution. Since the thread has no knowledge of OS scheduling decisions, it computes an inaccurate (de)registration condition. The conservative policy does not have these challenges, but cannot save as much memory when they are not relevant, as with applications that are not communication intensive. Since memory registration sizes are likely to be more significant for communication intensive applications, we favor the conservative policy.

## 6. Performance

We implement our memory registration scheme in Open MPI 1.4 and evaluate its performance with microbenchmarks and the NAS parallel benchmarks. Our evaluation system has 320 nodes, each with two 2.8GHz Intel Xeon quad-core processors, 8GB of physical memory and a Mellanox Infiniband ConnectX HCA. Each processor has two 6MB L2 caches (two cores share each L2). Each core has a 64KB L1 cache. Our registration scheme provides a helper thread per MPI task. The helper thread shares the L2 cache with the associated task. In particular, the task and helper thread share the 5.6KB DQ data structure, which easily fits in the L2 cache.

### 6.1 Microbenchmark Tests

Our first set of experiments uses the benchmark that Figure 1 shows. We compare the performance of the memory registration scheme using helper threads, the Open MPI 1.4 default scheme and


(a) Registered memory size (4MB messages)

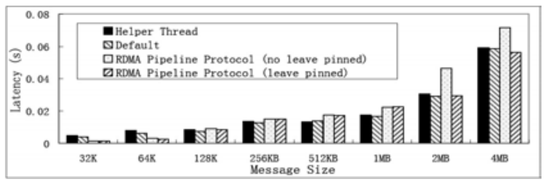
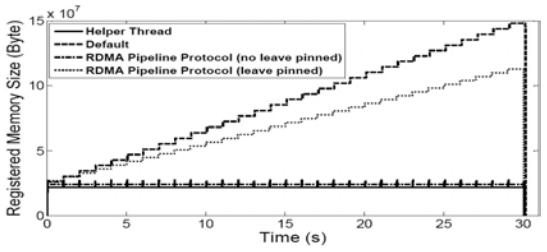

(b) Message latency

Figure 9: Buffer reuse tests


(a) Registered memory size (4MB messages)

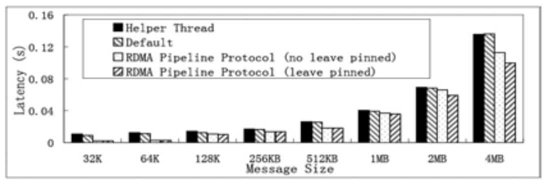

(b) Message latency

Figure 10: No buffer reuse tests

the RDMA pipeline protocol [33]. The Open MPI 1.4 default setting registers the memory on demand and leaves registered memory pinned for future use (our charts label this scheme "Default"), which many communication libraries use. The RDMA pipeline protocol represents a technique to reduce memory registration overhead. We leave memory both pinned and unpinned with the pipeline protocol to capture its full potential range. We conduct tests with both buffer reuse and no buffer reuse. In these tests "buffer reuse" means that we use the same user buffer for communication across loop iterations; "no buffer reuse" means the opposite. We report the average value of 100 runs of each test.

Figure 9 depicts the results for the buffer reuse tests. Our helper thread approach performs similarly to Default, which demonstrates the low helper thread overhead. With message sizes larger than 1MB, the overhead is less than 5%. Also, the helper thread scheme reduces the registered memory size compared to Default by 24.75%. The pipeline protocol implementation outperforms the other schemes for message sizes less than 256KB. Open MPI does not register memory for these messages with this protocol. Instead, it uses the bounce buffer and avoids registration and deregistration costs. Default and our helper thread scheme do not perform as well because they search a memory registration cache to find preregistered memory locations and have a longer function call chain. However, these schemes outperform the pipeline protocol for message sizes of 512KB and 1MB, despite the latter still using bounce buffers, which incurs data copying overhead. The pipeline protocol registers memory regions for larger message sizes. The pipeline protocol has low performance if it does not leave memory pinned since it repeatedly pays registration and deregistration costs. However, the pipeline protocol does not clearly benefit from its memory registration optimization if it leaves memory pinned. In terms of registered memory size, the helper thread saves more memory than the pipeline protocol (by 3.84% for no leave pinned and 21.04% for leave pinned). The pipeline protocol (leave pinned) uses less registered memory than Default and uses more registered memory than the helper thread, because the pipeline protocol registers an extra bounce buffer that it leaves pinned to transfer a message data segment for the first communication. This buffer reduces the registered memory for later communication.

Figure 10 depicts the results for the no buffer reuse tests. Figure 10 shows that the registered memory size for Default grows throughout the no buffer reuse test. The peak registered memory is 4.62× more than with the helper thread scheme, while the two schemes have comparable performance. Also, the helper thread scheme and Default both register memory on the critical path since the memory access pattern is random and, therefore, irregular. Their performance is worse than the pipeline protocol, which demonstrates that the pipeline protocol effectively reduces registration costs. However, the pipeline protocol (no leave pinned and leave pinned) uses more registered memory (3.84% and 3.31× more respectively) than the helper thread scheme. The pipeline protocol (leave pinned) uses less registered memory than Default, due to the bounce buffer mentioned above.

To summarize, the helper thread scheme has the minimum registered memory size in all cases. With predictable memory usage, as HPC applications often exhibit, the helper thread performs comparably to the scheme with the best performance for large messages. Even with irregular memory usage, the helper thread achieves performance similar to the Default scheme typically used in most communication libraries.

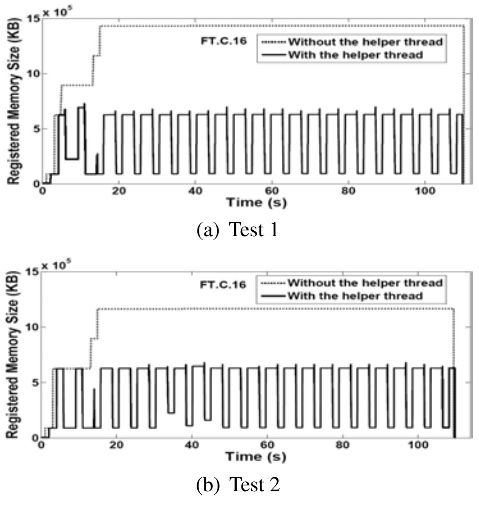
### 6.2 NAS Parallel Benchmarks

We apply the helper thread scheme for memory registration to the NAS parallel benchmarks. Figure 11 shows registered memory variance from MPI_Init to MPI_Finalize for Leave Pinned and our helper thread scheme. We only show two time steps or two iterations of the main computation loop (except FT with the 1D processor layout, where we show all iterations) to display the variance clearly. The other time steps or computation iterations have similar variance. We run one MPI task per node with four nodes total.

Our helper thread scheme reduces registered memory by 16.13%, 18.43% and 8.68% for BT, LU and SP. These benchmarks have communication phases before/after long computation phases, which exhibit repeating patterns that the registered memory variance clearly shows. Our predictor captures these patterns and directs the helper thread to register/deregister memory on time, which reduces the peak registered memory size. CG and FT are both communication intensive applications that interleave computation with frequent communication. The helper thread scheme does not reduce the peak registered memory size for CG, due to its communication intensity. However, it does reduce the average registered memory size compared to Leave Pinned. Collective operations consume significant registered memory with FT, for which we reduce the peak registered memory size by 31.0%. We incur negligible performance loss or marginal performance gain for all benchmarks (Table 2). Proactive memory deregistration (instead of leaving memory regis-

| | BT.D.4 | LU.D.4 | SP.D.4 | CG.D.4 | FT.B.4 |
|---|---|---|---|---|---|
| Helper thread | 265.98 | 219.39 | 140.59 | 200.04 | 79.47 |
| Leave Pinned | 266.87 | 218.80 | 141.10 | 202.31 | 79.50 |

Table 2: Execution time (s) for 5 NAS benchmarks.



(a) Test 1

(b) Test 2

Figure 12: Task aggregation tests

| | Test 1 | Test 2 |
|---|---|---|
| Helper thread | 102.62 | 102.64 |
| Leave Pinned | 104.05 | 103.36 |

Table 3: Execution time (s) for FT.C.16 aggregation tests

tered until MPI_Finalize) slightly improves performance (1.12%) for CG. On a system with a higher deregistration cost, such as Myrinet/GM [7], we expect a larger performance improvement.

Figure 11(f) displays the scenario we have shown in section 1. Without the helper thread, the application cannot continue execution since a single task pins 3.58GB memory; with the helper thread, we can constrain the registered memory size beneath the memory registration limit so that the application completes.

### 6.3 Task Aggregation Tests

This section examines the impact of dynamic memory registration via helper threads when aggregating multiple MPI tasks per node to save power or to improve performance [20]. We conduct experiments with four tasks per node that we randomly group before assigning them to nodes. Figure 12 depicts the performance of FT, problem size C with a 2×8 2D processor layout. We compare the performance of our helper thread scheme to Leave Pinned. We report the registered memory size per node from two tests with different random task groupings.

The helper thread approach reduces registered memory size significantly (49.39% and 41.70%), while also improving performance slightly (Table 3). We also find that the registered memory size between the two tests varies substantially with "Leave Pinned", while it is consistent with helper threads. Further analysis reveals that different task groupings change the characteristics of internode communication. Registered memory size varies substantially with "Leave Pinned" when the volume and frequency of internode communication changes due to poor task placement. By managing memory through helper threads, we can reduce registered memory independently of how tasks are grouped.

| | Strong Scaling (node number) | | |
|---|---|---|---|
| | 16 | 32 | 64 |
| Helper thread | 2082.73 | 1055.95 | 519.57 |
| Leave Pinned | 2085.86 | 1056.00 | 518.53 |

| | Weak Scaling (node number) | | |
|---|---|---|---|
| | 16 | 32 | 64 |
| Helper thread | 1041.58 | 1055.95 | 1093.16 |
| Leave pinned | 1039.46 | 1056.00 | 1092.23 |

Table 4: Execution time (s) of FT scaling tests.

### 6.4 Scaling Tests

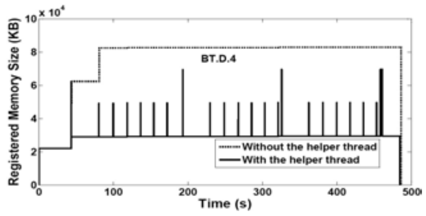
We extend our analysis to larger system scales to investigate how our memory registration scheme behaves as the node count changes. Figure 13 displays the results of FT with a 1D layout and one task per node under strong scaling (maintaining the same total problem size for all node counts). Although the registered memory size decreases as we increase the process count, as expected, our memory registration scheme achieves proportional memory savings in all cases. We save 33.19% for 16 nodes, 34.08% for 32 nodes and 31.70% for 64 nodes. Figure 14 displays results with weak scaling, which adjusts the input problem size to keep the workload per task approximately constant. Unlike the strong scaling tests, the weak scaling tests have an almost constant registered memory size as we increase the process count. We save 33.05%, 34.08%, and 33.09% for 16 nodes, 32 nodes and 64 nodes. Because the workload per task is approximately stable, each task has approximately the same amount of data to communicate across all scales. In all cases, our scheme reduces registered memory size at slightly better performance (Table 4).

## 7. Related Work

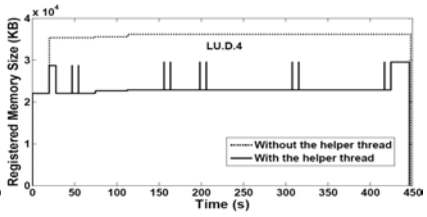
**Exploring the Predictability of HPC Applications:** Many prior efforts have studied common communication patterns in HPC applications [5, 15, 34]. HPC applications often exhibit spatial locality [15] (i.e., a small number of communication partners) and temporal locality [10] (i.e., iterative patterns). Researchers have proposed prediction techniques to leverage these characteristics to optimize communication performance. Afsahi et al. [2] propose several heuristics to predict MPI activity. They use these heuristics to cache incoming messages before posting the corresponding receive calls. Freitag et al. [9, 10] propose a periodicity-based predictor based on the computation of distances between patterns in data streams. They apply the predictor to prepare the receiver memory to guarantee enough resources for message reception. Other work leverages time delay neural networks and Markov models [27] to detect memory access patterns. These methods tend to require long training times. Ratn et al. [26] enable timed replay of communication events by distinguishing histograms with call stack and path sequence. Their approach does not consider the runtime overhead due to the nature of the approach's performance analysis, while the overhead can be comparable to communication time.

Our work has two primary differences from prior work on communication predictability. First, we use a time-based predictor. Time information is important for prediction-based memory management in order to meet registration deadlines while minimizing registered memory sizes. Prior work focuses on iterative memory access patterns while ignoring time information, thus missing opportunities to reduce registered memory sizes. Second, our predictor is lightweight and has a short training process, which makes it appropriate for online optimization.

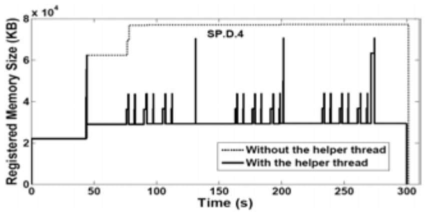
**Memory Registration Optimization:** Woodall et al. [33] described a pipeline protocol to overlap memory registration with RDMA operations. Their work changes the RDMA communication protocol, while our work is independent of the RDMA communi-



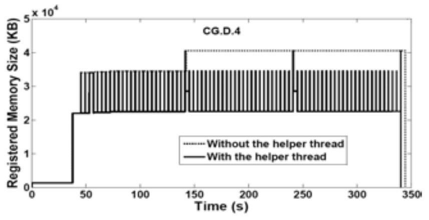
(a) BT



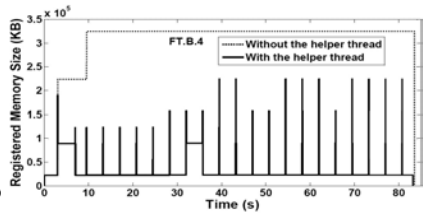
(b) LU



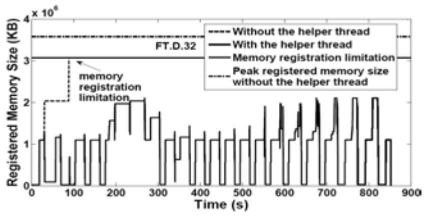
(c) SP



(d) CG

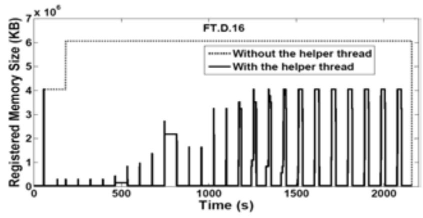


(e) FT

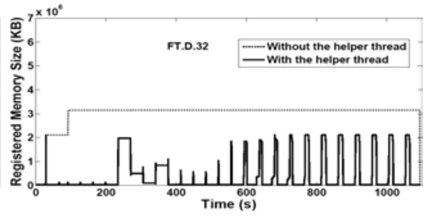


(f) FT with memory registration limitation

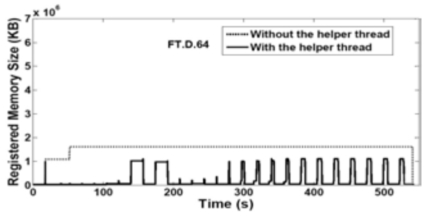
Figure 11: Memory registration sizes of the NAS parallel benchmarks



(a) 16 processes

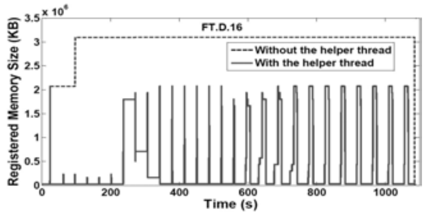


(b) 32 processes

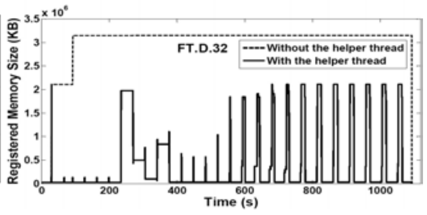


(c) 64 processes

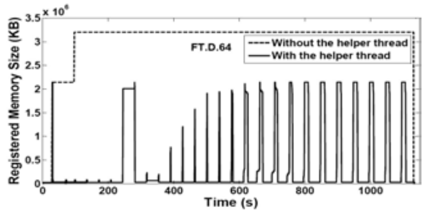
Figure 13: Strong scaling tests



(a) 16 processes



(b) 32 processes



(c) 64 processes

Figure 14: Weak scaling tests

cation implementation. Zhou et al. [35] combined memory pinning and allocation to eliminate pinning and unpinning from the registration and deregistration paths. They also batch deregistrations to reduce the average cost. Farreras et al. [7] proposed a pin-down cache for Myrinet. They delay deregistration and cache registration information for future accesses to the same memory region. Bell and Bonachea [3] proposed an RDMA algorithm for a shared memory system that determines the largest amount of memory that remote machines can share and then registers all shared memory regions and links them to a firehose interface, to which remote machines can write and read shared memory at any time. Unlike our scheme, the above work, when applied to reduce registered memory sizes, cannot completely remove registration and deregistration overhead from the communication critical path.

Shipman and Brightwell [28] investigated network buffer utilization and introduced a new protocol to increase the efficiency of receiver buffer utilization for Infiniband. The protocol uses buckets of receive buffers of different sizes, instead of consuming buffers regardless of the actual incoming message size. They target optimization of bounce buffers, while our work aims to reduce registered user buffer sizes. We register memory based on the real size of user buffers instead of a matched bucket size of network buffers.

**Using Helper Threads:** With the advent of multicore processors, several studies have proposed using idle cores to offload management tasks to helper threads. Tiwari et al. [32] perform dynamic memory allocation with a helper thread. They use prediction for memory preallocation of objects of a specific size. They only preallocate memory at the first allocation or after observing several allocation requests. Their prediction lacks time information and cannot be applied to minimize registered memory. Helper threads have also been used to perform branch prediction [23], prefetching [19] and adapting application execution to CPU availability [6]. We use a time-oriented design to direct helper thread operations, which distinguishes our contribution from prior work.

## 8. Conclusions

Efficiently managing the physical memory required for communication enables parallel application developers to solve larger problems in-core. This paper presented a novel scalable memory registration/deregistration scheme to reduce registered memory sizes for RDMA-based communication on HPC clusters based on multicore processors. We leverage a helper thread to register and to deregister user buffers according to a registration time prediction. Our approach reduces the amount of registered memory while avoiding memory registration/deregistraton overhead, a result that prior methods cannot achieve. We design a context-aware predictor to provide highly accurate time prediction. We presented a time-oriented helper thread design. Our design delays memory registration and avoids the accumulation of registered memory. We discussed design issues, such as synchronization, how to minimize the latency introduced to the communication critical path, and how to reduce the resource requirements of helper threads. We investigate how to distribute helper threads between limited idle cores to minimize registered memory size. We applied our framework to the NAS parallel benchmarks and reduced registered memory by 23.62% on average and up to 49.39%. Our mechanisms outperform existing memory efficient solutions and achieve similar performance while using much less memory than existing high performance solutions.

## Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the HiPEAC-2 (grant agreement no. 217068) and I-CORES (grant agreement no. 224759) projects. This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DEAC52-07NA27344 and supported by NSF (CNS-0905187, CNS-0910784, CCF-0848670, CNS-0709025, CNS-0720750).

## References

[1] Adaptive Computing Company. TORQUE Resource Manager. http://www.clusterresources.com/products/torque-resource-manager.php.

[2] A. Afsahi and N. J. Dimopoulos. Efficient Communication Using Message Prediction for Cluster Multiprocessors. In *Proceedings of International Workshop on Network-Based Parallel Computing*, Toulouse, France, 2000.

[3] C. Bell and D. Bonachea. A New RDMA Registration Strategy for Pinning-Based High Performance Networks. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Nice, France, 2003.

[4] 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. In *IEEE/ACM Micro*, 1995.

[5] R. Brightwell, S. Goudy, A. Rodrigues, and K. Underwood. Implications of Application Usage Characteristics for Collective Communication Offload. *International Journal of High-Performance Computing and Networking*, 4:104–116, 2006.

[6] Y. Ding, M. Kandemir, P. Raghavan, and M. Irwin. A Helper Thread Based EDP Reduction Scheme for Adapting Application Execution in CMPs. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Miami, USA, 2008.

[7] M. Farreras, G. Almasi, C. Cascaval, and T. Cortes. Scalable RDMA Performance in PGAS Languages. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, 2009.

[8] F. Freitag, J. Caubet, M. Farrera, T. Cortes, and J. Labarta. Exploring the Predictability of MPI Messages. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Nice, France, 2003.

[9] F. Freitag, J. Corbalan, and J. Labarta. A Dynamic Predictor Detector: Application to Speedup Computation. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, San Francisco, USA, 2001.

[10] F. Freitag, M. Farrera, T. Cortes, and J. Labarta. Predicting MPI Buffer Addresses. In *Proceeding of International Conference on Computational Science*, Krakow, Poland, 2004.

[11] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, and J. M. S. etc. al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Euro PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

[12] W. Huang, G. Santhanaraman, H. Jin, Q. Gao, and D. K. Panda. Design and Implementation of High Performance MVAPICH2: MPI2 over Infiniband. In *Proceedings of IEEE International Symposium on Cluster Computing and Grid*, Singapore, 2006.

[13] Infiniband Trade Association. Infiniband Roadmap. http://www.infinibandta.org.

[14] Y. Iwamoto, K. Suga, K. Ootsu, T. Yokota, and T. Baba. Receiving Message Prediction Method. *Parallel Computing*, 29:1509–1538, 2003.

[15] J. Kim and D. Lilja. Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs. In *Network-Based Parallel Computing Communication, Architecture, and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[16] M. Koop, P. Shamis, I. Rabinovitz, and D. K. Panda. Designing High Performance and Resilient Message Passing on Infiniband. In *Proceedings of Workshop on Communication Architecture for Clusters*, Atlanta, USA, 2010.

[17] Lawrence Livermore National Lab. ASC Sequoia Benchmarks. https://asc.llnl.gov/sequoia/benchmarks.

[18] Lawrence Livermore National Laboratory. SLURM: A Highly Scalable Resource Manager. http://https://computing.llnl.gov/linux/slurm.

[19] J. Lee, C. Jung, D. Lim, and Y. Solihin. Prefetching with Helper Threads for Loosely-Coupled Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 20:1309–1324, 2009.

[20] D. Li, D. S. Nikolopoulos, K. W. Cameron, B. R. de Supinski, and M. Schulz. Power-Aware MPI Task Aggregation Prediction for High-End Computing Systems. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Atlanta, USA, 2010.

[21] F. Mietke, R. Baumgartl, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In *Proceedings of EURO-PAR*, Dresden, German, 2006.

[22] MPICH2. A High-Performance and Widely Portable Implementation of MPI Standard. http://www.mcs.anl.gov/research/projects/mpich2.

[23] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Effective Alternative to Large Instruction Windows. *IEEE Micro*, 23:20–25, 2003.

[24] L. Ou, X. He, and J. Han. An Efficient Design for Fast Memory Registration in RDMA. *Journal of Network and Comptuer Applications*, 32:642–651, 2009.

[25] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22:46–57, 2002.

[26] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz. Preserving Time in Large-Scale Communication Traces. In *Proceedings of International Conference on Supercomputing*, Island of Kos, Greece, 2008.

[27] M. F. Sakr, S. P. Levitan, D. M. Chiarulli, B. G. Horne, and C. L. Giles. Predicting Multiprocessor Memory Access Patterns with Learning Models. In *Proceedings of International Conference on Machine Learning*, Nashville, USA, 1997.

[28] G. Shipman, R. Brightwell, B. Barrett, J. M. Squyres, and G. Bloch. Investigations on InfiniBand: Efficient Network Buffer Utilization at Scale. In *European PVM/MPI Users' Group Meeting*, Paris, France, 2007.

[29] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges. InfiniBand Scalability in Open MPI. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, 2006.

[30] H. Subramoni, M. Koop, and D. K. Panda. Design Next Generation Clusters: Evaluation of Infiniband DDR/QDR on Intel Computing Platforms. In *Proceedings of High Performance Interconnects*, New York, USA, 2009.

[31] S. Sur, A. Vishnu, H.-W. Jin, W. Huang, and D. K. Panda. Can Memory-less Network Adapter Benefit Next-Generation Infiniband Systems? In *13th Symposium on High Performance Interconnects*, Stanford, USA, 2005.

[32] D. Tiwari, S. Lee, J. Tuck, and Y. Solihin. MMT: Exploiting Fine-Grained Parallelism in Dynamic Memory Management. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Atlanta, USA, 2010.

[33] T. S. Woodall, G. M. Shipman, G. Bosilca, and A. B. Maccabe. High Performance RDMA Protocols in HPC. In *Euro PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Bonn, Germany, 2006. Springer-Verlag.

[34] R. Zamani and A. Afsahi. Communication Charateristics of Message-Passing Scientific and Engineering Applications. In *Proceedings of Parallel and Distributed Computing and Systems*, Phoenix, USA, 2005.

[35] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubincki, J. Philibin, and K. Li. Experiecne with VI Communication for Database Storage. In *Proceedings of International Symposium on Computer Architecture*, Anchorage, USA, 2002.