RDMA Service using Dynamic Page Pinning: An Onloading Approach

K. VAIDYANATHAN, M. SCHLANSKER, J. MUDIGONDA, N. BINKERT AND D. K. PANDA

Technical Report
Ohio State University (OSU-CISRC-5/08-TR21)

RDMA Service using Dynamic Page Pinning: An Onloading Approach

K. Vaidyanathan
Comp. Science and Engg.,
Ohio State University
vaidyana@cse.ohio-state.edu

M. Schlansker, J. Mudigonda, N. Binkert
Advanced Architecture Lab
HP Laboratories
{mike_schlansker,jayaram.mudigonda,nathan.binkert}@hp.com

D. K. Panda
Comp. Science and Engg.,
Ohio State University
panda@cse.ohio-state.edu

Abstract

Remote Direct Memory Access (RDMA) has proven beneficial to high-performance computing tasks and shows great promise for commercial computing tasks such as network-based file systems, distributed databases, and remote procedure calls. While existing RDMA implementations do provide efficient one-sided, inter-node remote memory access, they do not preserve all of the benefits of virtual memory. Further, RDMA memory regions are either managed directly by users (requiring user control over system critical resources) or by using complex NIC hardware. We address these limitations by using a software-centric onloading approach and demonstrate it on a Ethernet platform. Our approach exploits the abundant compute cycles in future many-core processors to perform tasks such as page pinning, DMA startup/completion, page releasing, flow control and page fault handling. Our architecture provides access to objects in unpinned virtual memory, simplifies the NIC hardware, and ensures robust system behavior by managing memory within trusted kernel code.

1 Introduction

Many high-performance networks such as InfiniBand [5], 10-Gigabit Ethernet [1], Quadrics [16], Myrinet [7] and JNIC [18] support Remote Direct Memory Access (RDMA) [3] to provide high-performance and scalability to applications. While there are multiple RDMA standards, in this paper, we use the generic term to denote one-sided inter-node memory access. Unlike two-sided sends and receives, one-sided operations access remote memory without requiring the remote application's participation. RDMA often combines one-sided execution with OS-bypass to achieve low latency and high bandwidth. This provides network-centric application primitives that achieve three major objectives: data is delivered without expensive software-based copies; concurrency is improved when one-sided access occurs without remote application cooperation; application processing is reduced when asynchronous remote processing is moved away from the application. RDMA defines two primitives: a *put* writes to, and a *get* reads from remote memory.

In the high-performance computing domain, the utility of RDMA is already proven [12, 13]. MPI is the most popular standard for parallel computing and networks such as InfiniBand, Quadrics, etc., use RDMA to accelerate MPI's two-sided messaging. More recently, MPI also includes direct support for one-sided communications that exploits optimized RDMA. RDMA's get and put operations are natural communication primitives for Partitioned Global Address Space (PGAS) Languages, such as Unified Parallel C (UPC) [2] and Co-Array Fortran [14], that incorporate benefits from message passing's scalability and shared memory's ease-of-programming.

RDMA also offers potential in data-center environments [19]. The distributed applications [20] hosted in these environments such as web and application servers, file systems, caching and resource management services can significantly benefit from RDMA for achieving data-center-wide scalability. Researchers have proposed several lower-level mechanisms such as Sinfonia [4], Khazana [8], DDSS [21] to build efficient data-center subsystems including cluster file system, distributed lock manager, and databases. These mechanisms typically deal with memory-based objects and manipulate these objects frequently. Thus, it is important to provide efficient distributed manipulation of memory-based objects using *get* and *put* operations to increase the performance and scalability.

While existing Remote Direct Memory Access (RDMA) provides a foundation, a closer inspection indicates that today's RDMA is not suitable for many of these environments. Firstly, existing RDMA implementations do not preserve all of the benefits of virtual memory to applications such as the illusion of using more memory than that is physically present and the protection capabilities for memory regions that are shared among user programs. Secondly, the memory regions used for RDMA are typically managed by users in an independent manner. Multiple users making in-

dependent decisions can lead to starvation of resources and robustness issues (e.g., a system crash due to unavailable pages). Networks such as Quadrics address some of these limitations by using complex NIC hardware that maintains the page tables and frequently interacts with the operating system. In this paper, we address these limitations by proposing a software-centric onload approach. The salient features and main contributions of the proposed approach are:

1. Our approach exploits the abundant compute cycles in future many-core processors to perform RDMA tasks. Our experimental results are measured on a working prototype and demonstrate a low-overhead for performing needed operations in the critical path.

2. Unlike many existing networks, our design preserves the key capabilities that are provided by virtual memory. The design allows access to more virtual memory than is physically present and supports access protection for client applications. This is especially important in complex multiple program environments associated with commercial computing.

3. Our design utilizes a kernel helper thread to manage memory pages leading to a robust and well controlled environment for managing the virtual memory subsystem. This compares to existing approaches which require that users manage memory pages or that pages are managed through complex NIC hardware.

4. Our design simplifies the NIC hardware by onloading RDMA tasks such as page pinning, DMA startup/completion, page unpinning, handling page faults and flow control to the kernel helper thread. Further, the presence of a kernel helper thread avoids replication of page table entries and provides faster access to page tables and helps in easier maintenance. In addition, there are no changes/updates required in NIC hardware as changes are made to the internal virtual memory subsystems.

Further, due to the presence of a kernel helper thread in our design, application-specific tasks such as distributed queue insertions/modifications/deletions, locking operations and several other memory-based operations can be onloaded, thus providing opportunities to revise the design and implementation of many subsystems in multi-program environments.

The rest of the paper is organized as follows. Section 2 provides a brief background on RDMA capabilities offered by existing high-performance networks, the JNIC architecture and the registration issues with RDMA. Detailed design of our RDMA prototype is discussed in Section 3. We evaluate our RDMA prototype and present the results in Section 4. In Section 5, we present the related work. We conclude the paper in Section 6.

2 Background and Motivation

In this section, we present the capabilities of RDMA operations in high-speed networks, the JNIC architecture and the registration issues with RDMA.

2.1 RDMA Operations in High-Speed Networks

Several high-speed networks such as InfiniBand, 10-Gigabit Ethernet, Quadrics, and Myrinet provide one-sided memory semantics communication model. Here, Remote Direct Memory Access (RDMA) operations are used which allow the initiating node to directly access the memory of remote-node without the involvement of the remote-side application. Hence, an RDMA operation has to specify both the memory address for the local buffer as well as that for the remote buffer.

2.2 JNIC Architecture

The Intel/HP Joint Network Interface Controller (JNIC) [18] prototype models in-data-center communications over Ethernet. Figure 1 shows a JNIC system consisting of front-side-bus-attached NIC hardware and optimized software using a dedicated kernel helper. The prototype hardware is an FPGA-based Gigabit Ethernet NIC that plugs into an Intel Xeon socket, allowing communication over the front side bus. A reliable communications (VNIC) layer implements JNIC-to-JNIC communication using the TCP protocol. The VNIC layer presents virtual NIC device interfaces to user or kernel tasks. Messages are sent by multiplexing message requests from VNIC clients to the reliable communications layer. Messages are received when the reliable communications layer receives a message and delivered to the appropriate receiving destination VNIC. The VNIC implements copy-free transport using physically addressed DMA. At the time of VNIC-layer registration, VNIC source and target buffers are physically locked/pinned regardless of when they are needed for future RDMA. For more details on JNIC architecture and VNIC layer, the readers are encouraged to refer to [18].

In this paper, we build on this prior work and describe the architecture of a working onload-style RDMA implementation. This architecture is carefully crafted to ensure forward progress even when client applications compete for limited resources. To support multiple client RDMA operations on regions of arbitrary size, a number of key tasks are executed. Tasks include: flow control, region segmentation, page pinning, initiating copy-free transport, and page unpinning.

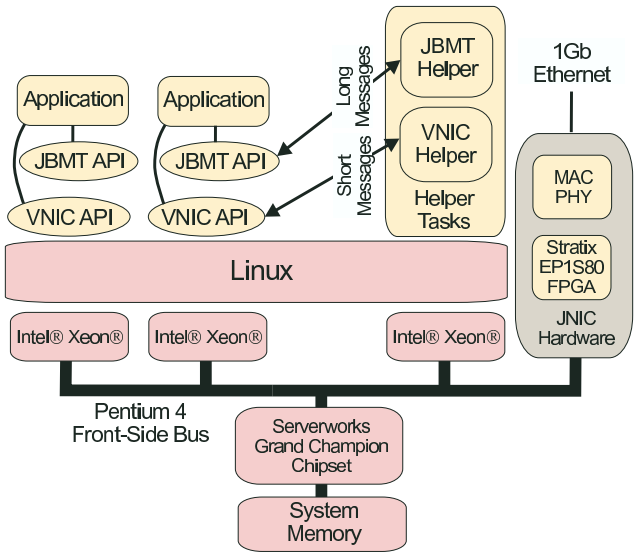

Figure 1. JNIC Prototype [18]

2.3 RDMA Registration

RDMA typically involves two steps: registration and the actual *put* or *get*. The registration process can be broadly classified into pinning-based registration [5, 1] and hardware-assisted registration [16, 15]. In this section, we present the details on these two registration strategies and its associated issues.

Pinning-based Registration: In this method, to register a buffer, a task makes a system call into a kernel component of the RDMA service. The kernel initializes the control data and creates a handle that encodes buffer access rights. The kernel then swaps in and pins (locks) all the buffer pages. After a successful buffer pinning, the kernel component asks the NIC to install a mapping between the handle and the pinned physical pages and waits for an acknowledgment. The buffer handle is then passed back to the user task after receiving the acknowledgment, which in turn sends the buffer handle to remote tasks to be used in subsequent *get* (or *put*) requests. Similarly, during a de-registration operation, the kernel component asks the NIC to remove the mapping between the handle and the pinned physical pages and waits for an acknowledgment. The kernel then unlocks all the buffer pages associated with the handle and removes the corresponding entries associated with the handle. Due to the page pinning restriction in the registration phase, this approach not only limits the amount of buffers registered to available physical memory but can also wastes the physical memory, if it is currently not utilized. Finally, allowing users to pin physical memory compromises robustness. One buggy application affects all others by monopolizing physical memory. Further, the cost of this registration and de-registration is typically expensive [11, 22] in networks such as InfiniBand.

Hardware-assisted Registration: In this approach (e.g. Quadrics [16]), the NIC combines a hardware TLB and tweaks in operating system's virtual memory support to allow the NIC to pin pages, initiate page faults and track changes in the application's page table. In this approach, there is no restriction on what memory is DMA-able, potentially the entire virtual memory can be made available for RDMA operations. However, pushing the responsibility of pinning and managing the page tables to the NIC comes with increased hardware cost and complexity [6, 23]. Further, the hardware requires frequent updates as changes are made to the internal virtual memory subsystems.

In the following sections, we describe an architecture that eliminates problems associated with user pinning and hardware-based registration by combining dynamic page pinning [10] (the ability to pin a small set of pages and make progress in a pipelined manner) and onloading [17] (the ability to perform the tasks on the host processor).

3 Proposed Design

In this section, we present the design goals and details of our proposed onloaded RDMA service.

Goals: To address the limitations mentioned in Section 2.3, our primary goal is to allow RDMA operations on unpinned virtual memory with simplified NIC hardware. Thus, we need a mechanism that supports page pinning only when necessary in a pipelined manner but at the same time, the pinning process should be rapid and should not slow down RDMA. In addition, we need a mechanism that jointly pins the pages on both sending and receiving side to allow a copy-free hardware transport and a mechanism that guarantees forward progress even when available memory is limited.

In the following sections, we present the detailed design of RDMA service that meets our design goals.

3.1 Basic Design

The basic idea of our design is to exploit the abundant compute cycles of future many-core processors to perform the tasks involved in *get* and *put* operations. We use a dedicated kernel helper thread to perform just-in-time physical page pinning, access rights enforcement, copy-free data transport, guaranteeing progress even when the pages are not resident in memory and flow control. We refer to the RDMA service as JNIC's Bulk Message Transport (JBMT) and present the details in performing a JBMT *get* operation in the rest of the paper.

Figure 2(a) shows the overall architecture of JBMT that provides copy-free autonomous message delivery. As

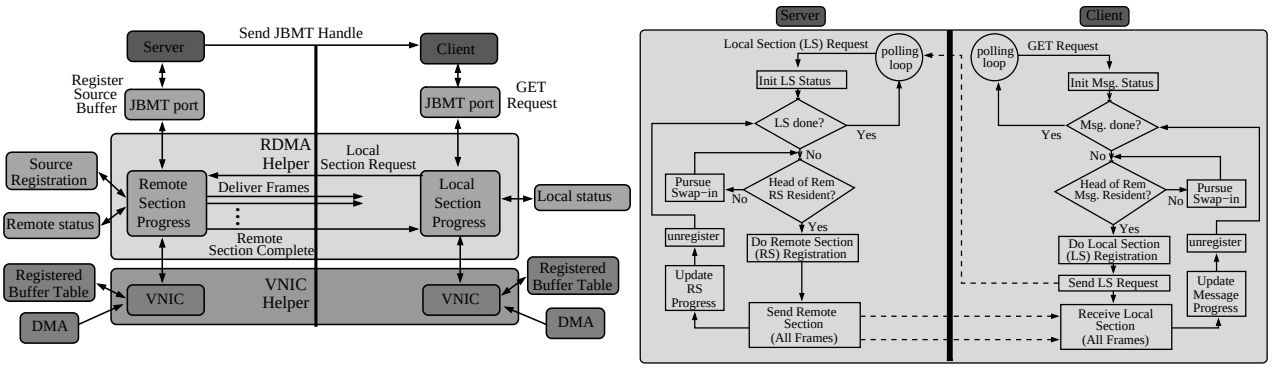

Figure 2. RDMA Service: (a) JBMT Architecture and (b) *get* Operation in JBMT

shown in the figure, the JBMT kernel helper thread receives requests and generates responses through virtualized JBMT command and completion ports. In this approach, a source buffer is registered by the server. JBMT registration provides access control for the source region and it does not pin the memory pages. As a result, it does not consume valuable resources which are otherwise required to manage the pinned memory regions. On a successful registration request, a token is generated by JBMT. The resulting registration token is sent to the receiver (client) using conventional VNIC messaging, which can be used for data transfer. During a get operation, the receiver specifies the source region token, an offset into the source region, a target region pointer, and a transfer length. Legal transfers are confined to source buffers for which access rights were granted in prior source buffer registrations.

After checking the access rights, JBMT decomposes a large message into smaller dynamically pinned sections. First, a local section (a small portion of a large *get* operation) is pinned. The local section can be any subset of the target region requested by the *get* operation as dictated by memory availability. A request for a local section is then sent to the server-side remote interface. After checking for available resources on the server, the server attempts to pin a remote section for data transfer. The remote section is again possibly a subset of the requested local section length. The kernel helper thread on the server attempts to transmit the remote section to the receiver as Ethernet-frame-size VNIC transfers. During this data transfer both the local and remote sections are pinned/locked and hence, we can directly perform DMA operations to transfer the data. After a remote section is transferred, it is unpinned and additional remote sections are pinned and transmitted until the entire requested local section has been received. After the local section is completed, it is unpinned and the next local section is pinned and the process continues until the entire buffer is transferred. Upon successful transfer of the entire buffer, a completion event is deposited in the JBMT port's completion queue, depending on application's request.

Figure 2(b) shows the detailed steps involved during a *get* operation. As shown in the figure, a *get* request is submitted as a message request into a local node command queue for processing by the dedicated JBMT kernel helper. This command queue (request and response) is a memory mapped queue that is shared between the application and the dedicated JBMT kernel helper thread. The helper thread detects the *get* request by polling on this command queue and the local target buffer handle is checked to ensure that virtual buffer access is valid. In JBMT registration, the buffers can be of arbitrary size and can exceed the size of physical memory. Hence, this requires that virtual user buffers are segmented into smaller pinned sections and processed sequentially. The *get* operation processes the target buffer handle to determine the appropriate target virtual address for data transfer. As mentioned above, the JBMT helper requests that the underlying VNIC layer register and pin a portion of a user buffer and return the successfully registered physical section size which depends on resource availability. This lower-layer registration pins sections of a larger virtual buffer temporarily and on demand. Further, if the pages are not resident in memory, the helper invokes a request to bring the swapped pages to memory and the details of how this is handled will be discussed in Section 3.4. Allowing the lower layer to determine the extent of physical pinning facilitates rapid progress when resources are plentiful and ensures forward progress when resources are scarce. When resources are plentiful, a large physical section minimizes pinning costs. When resources are scarce, progress is ensured when only the head of the buffer is physically registered and as little as a single page is pinned.

Once the maximum size of the local section is determined, a local section request is sent to the server-side remote interface and similar steps are followed on server-side to determine the minimum remote section. This architecture is carefully crafted to ensure forward progress on both local and remote nodes. Upon completion of remote and

local sections, the status of the corresponding request is updated, the locked pages are unpinned and proceeds to the next local section.

In the following sections, we present the details of how we handle page pinning while guaranteeing progress when pages are not in memory, flow control and page swapping.

3.2 Handling Page Pinning

Page pinning consists of two actions. First, pinning requires that the head of a requested buffer is physically resident. Thus, if the request buffer pages are not physically resident, JBMT kernel helper thread stimulates stimulates the swap-in of needed pages. Second, page pinning prevents future swap out of memory pages by marking kernel tables. When pages are missing, a separate thread of execution swaps needed pages. When no pages are resident, the resulting physical section has size zero, and DMA is delayed. JBMT keeps track of the pending RDMA tasks and periodically attempts to lock a non-empty physical section, if the head of the particular section is currently resident in memory.

3.3 Handling Flow Control

Though the actual data is delivered through DMA, JBMT sends control commands to remote-node to perform the appropriate DMA data transfers. Thus, JBMT layer needs to ensure that there is space on the remote-side to accept control commands and thus requires flow-control mechanisms to prevent scenarios such as too many JBMT requests from a single receiver, too many local section requests and several others. As shown in Figure 3, we use a credit-based flow control in our design. Every JBMT *get* operation reserves a local credit in order to send a local section request (shown as Step 1 in the figure). Further, it also reserves enough control credits for the remote node to send the control commands in performing the DMA transfer for the requested local section. After the needed credits are successfully acquired, the client-side sends a local section request to the remote node. Similarly, the remote node attempts to acquire a remote section credit. This limit is applied to prevent a remote section to service a large number of local section requests. During the failure of this event, the remote-node sends appropriate NACK messages to release the credits/pages on the local-side and repeat Step 1 at a later point in time using a timer. However, if enough remote section credits are available, the remote node starts delivering the frames to the local node. After completion, the local and remote node releases the credits to process future local and remote section requests.

3.4 Handling Page Swapping

In operating systems like Linux, the kernel helper function that helps to lock/pin the user pages is typically a blocking call. In other words, if the user pages are not resident in memory, the helper function usually does not return until the pages are swapped in and a lock is acquired on these pages. As a result, the dedicated kernel helper thread may block for a very long time if it attempts to directly lock buffers that are currently not resident in memory. This would also result in blocking JBMT operations submitted by other tasks that are currently in-flight and would significantly affect the performance of any other JBMT client tasks. To avoid this scenario, we use an asynchronous page fault handler thread to handle page fault requests from JBMT kernel helper thread. The basic design of this thread is to accept a sequence of page fault requests and make progress on these requests. This thread attempts to bring in pages from disk for a small portion of the accessed user buffer. The thread only touches the pages and does not pin or lock the pages in physical memory during a page fault miss. This can also modified so that the first page of the buffer can be locked/pinned and the remaining pages can just be swapped in. Since the JBMT helper thread periodically checks if pages are resident in memory and immediately locks the pages if it is, locking the first page may not be necessary. Also, to process multiple page fault requests at the same time, we spawn several asynchronous page fault handler threads and the JBMT service chooses these threads in a round-robin manner to submit page fault requests.

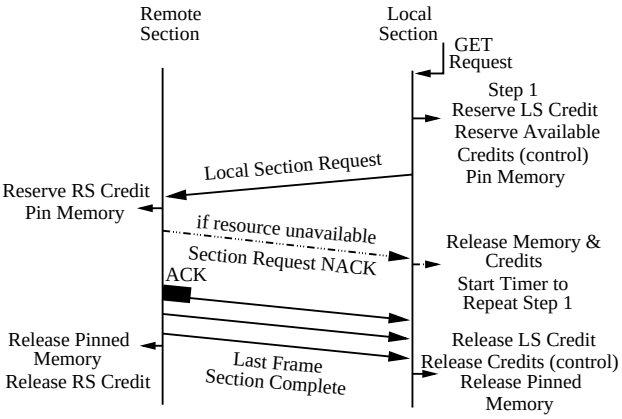

Figure 3. Flow Control in JBMT

To summarize, our architecture provides substantial enhancement to traditional RDMA by onloading the RDMA tasks such as page pinning, DMA startup/completion, flow control, etc. Our architecture provides unlimited access to objects in unpinned virtual memory, simplifies the NIC hardware and supports more control in managing memory pages.

4 Experimental Results

In this section, we analyze the performance of our proposed RDMA service. Our experimental test bed consists of two nodes with two 3 GHz Xeon "Gallatin" processors with a 512 KB cache and 512 MB memory. The FPGA-based 1-Gigabit Ethernet NIC is connected via the FSB. In our prototype implementation, we design the *put* as a remote *get* operation and defer a more efficient *put* implementation for future work.

4.1 Latency

In this experiment, we show the latency of a JBMT *get* operation. We perform the benchmark in the following way. To measure the *get* latency, we initiate a JBMT *get* operation and poll on the memory to wait for completion. After completion, we post the next JBMT *get* operation. We repeat this for several iterations and report the average, as shown in Figure 4. As mentioned earlier, a JBMT *get* operation involves processing local and remote sections (shown as JBMT Processing Time in the figure), a local section request (shown as VNIC Control Message Time in the figure) and the actual data transfer (shown as VNIC Data Transfer Time in the figure). We see that the latency of *get* operation for a 1 byte message is 19 μs. Further, we observe that the *JBMT Processing Time* and the VNIC Control Message Time occupies only 3 μs and 7 μs, respectively. Also, we see that the *JBMT Processing Time* does not increase with increasing message sizes. However, *JBMT Processing Time* will start varying when the buffers span over multiple pages and multiple local and remote section sizes, especially for very large *get* operations. There are existing caching techniques [22, 15] which can be used to further alleviate this overhead.

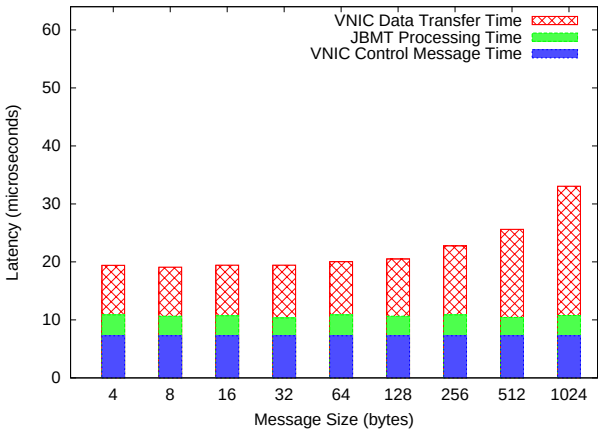

Figure 4. Latency of *get* operation

4.2 Bandwidth

Here, we present the bandwidth performance of the *get* operation. To measure the *get* bandwidth, we post a window of *get* operations. After every *get* completion, we post another *get* operation and repeat this for several iterations and measure the *get* bandwidth. Figure 5 shows the bandwidth performance of *get* operation. We see that the JBMT *get* can achieve a peak bandwidth of up to 112 MB/s for very large messages, thus almost saturating the link bandwidth. Hence, it demonstrates that performing page pinning during a JBMT *get* operation does not significantly affect the bandwidth performance. However, for very small messages, we see that the JBMT *get* shows poor bandwidth due to several factors including latencies required for Ethernet transmission, needed page pinning in the critical path, and limitations of our prototype.

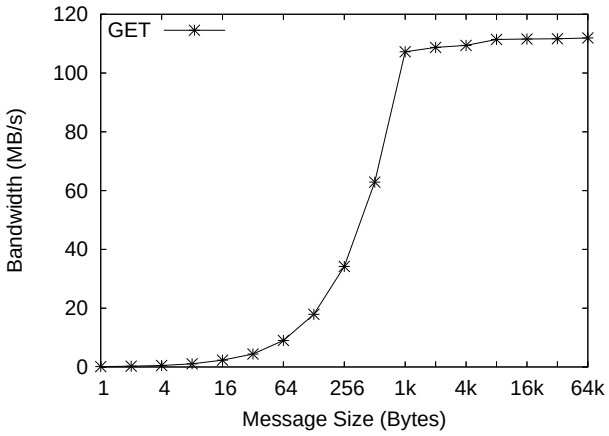

Figure 5. Bandwidth of *get* operation

4.3 Registration Cost

In this experiment, we measure the registration and de-registration cost in JBMT. This registration is different from the VNIC-layer registration which pins/locks the memory pages. JBMT registrations do not pin any pages. It only creates a local handle which can be used by peer nodes for a future JBMT operation. We perform several JBMT registrations of a particular message size and report the average latency in performing the JBMT registration. The JBMT de-registration cost is measured in a similar way. Table 1 reports the cost of registration and de-registration operations in JBMT. Both registration and de-registration costs remain constant irrespective of the message size of less than 2 μs. Due to the fact that pages are not pinned and no page translations are maintained in the NIC, the registration and de-registration operations remain constant and inexpensive.

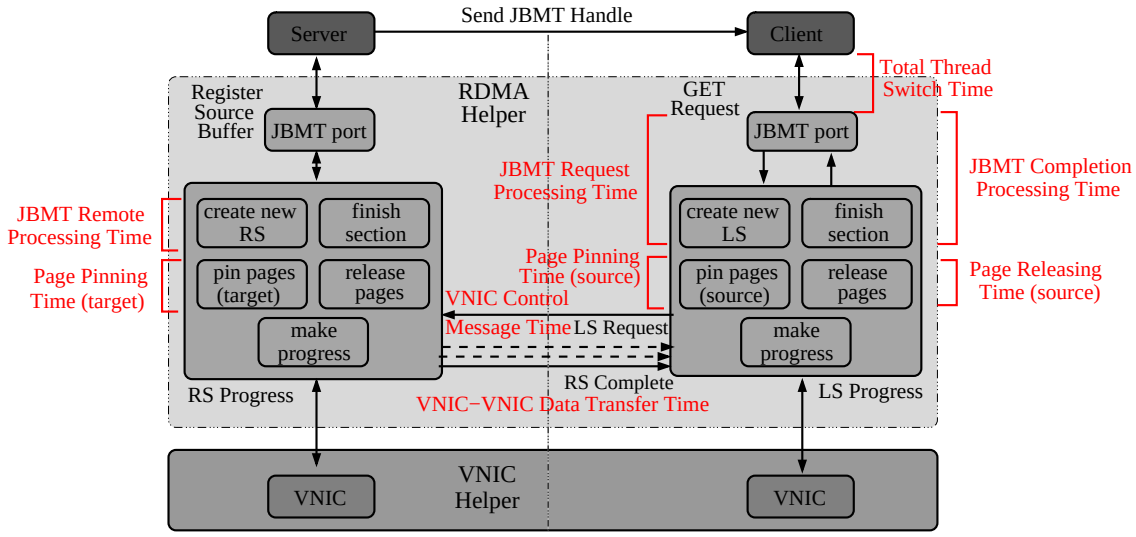

Figure 6. Timing Measurements of JBMT *get*

Table 1. Registration Cost

	Registration (usecs)	De-Registration (usecs)
Any Msg. Size	1.32	0.25

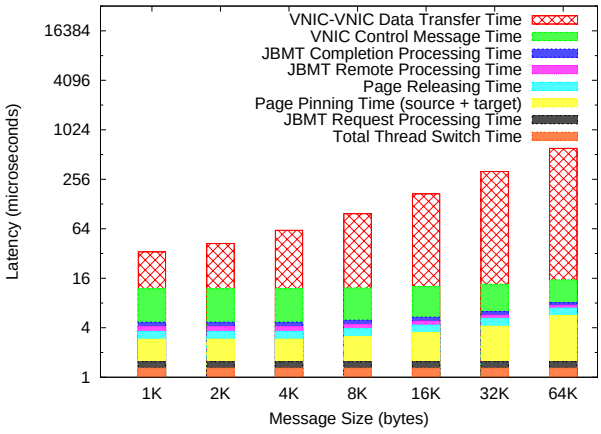

Figure 7. Cost Breakdown of JBMT *get*

4.4 Cost breakdown of JBMT

To further analyze the JBMT *get* operation in detail, we measure the cost of several steps involved in JBMT *get* operation and report its overhead in Figures 6 and 7. The detailed tasks during a *get* operation (shown in red color) is shown in Figure 6. The *Total Thread Switch Time* indicates the time taken to switch from the application that initiates the *get* operation to the JBMT kernel helper thread that listens for such requests. The *JBMT Request Processing Time* refers to the time spent before initiating a local section request and the *JBMT Completion Processing Time* refers to the time spent after receiving all the remote frames and the post processing of a local section. The *Page Pinning Time* and *Page Release Time* refer to the time spent in kernel for locking and releasing the user pages. The *VNIC Control Message Time* refers to the time spent for sending a local section request to the remote side. The *JBMT Remote Processing Time* refers to the time spent by the remote side in initiating the remote section frames. The *VNIC-VNIC Data Transfer Time* indicates the time spent in sending the data using the underlying VNIC layer.

In Figure 7, we observe the time spent by each of these operations for various message sizes. We see that the time spent by *Total Thread Switch Time*, *JBMT Request, Remote and Completion Processing Time* is much less when compared to other components in a JBMT *get* operation. However, as mentioned earlier, due to our prototype hardware, the time spent by these operations is still considered quite high and there is room for improving this further. Further, we see that *Page Pinning Time*, *Page Release Time* and *VNIC-VNIC Data Transfer Time* increases with increasing message sizes. As message size increases, the buffers span

multiple pages which automatically increases the page pinning/unpinning costs. Also, since the message sizes are less than the maximum allowed local and remote section sizes (1 MB), we observe that the overhead of *JBMT Request Processing Time*, *JBMT Completion Processing Time* and *JBMT Remote Processing Time* does not increase with increasing message size. The overhead of these operations is expected to increase as the number of local/remote sections increases. However, this overhead will be significantly less than the overall time taken to perform the *get* operation.

5 Related Work

Modern processors are seeing a steep increase in the number of cores available in the system [9]. As the number of cores increases, the choice to dedicate one or more cores to perform specialized functions will become more common. In this paper, we proposed a mechanism to onload the RDMA tasks to a dedicated kernel helper thread for such systems. Researchers [5, 16, 1, 15, 10, 17] have proposed several mechanisms in designing RDMA operations and onloading techniques. In this work, we combined the existing dynamic page pinning and onloading technique to perform RDMA related tasks. Unlike conventional RDMA, buffer pinning is kernel-managed to allow the system to have better control over critical resources in our proposed design. Our architecture preserves the benefits of virtual memory in a robust and well controlled environment. Further, the presence of a kernel helper thread avoids replication of page table entries and provides faster access to kernel page tables and page pinning, and simplifies the NIC hardware as compared to existing approaches.

6 Conclusions and Future Work

While existing RDMA implementations do provide efficient one-sided, inter-node remote memory access, they do not preserve all of the benefits of virtual memory. Further, RDMA memory regions are either managed directly by users (requiring user control over system critical resources) or by using complex NIC hardware. In this paper, we addressed these limitations by using a software-centric onloading approach to perform tasks such as page pinning, DMA startup/completion, page releasing, flow control and page fault handling. Our architecture provides access to objects in unpinned virtual memory, simplifies the NIC hardware, and ensures robust system behavior by managing memory within trusted kernel code. As a part of future work, we propose to perform application-driven evaluations and further enhance the performance of get and put operations.

This paper is based on research carried out by Karthikeyan Vaidyanathan during his summer internship at HP. The OSU part of the research is supported in part by NSF grants #CNS-0403342 and #CCF-0702675.

References

[1] 10 Gigabit Ethernet Alliance. http://www.ethernetalliance.org/.

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

[3] RDMA Consortium. http://www.rdmaconsortium.org/home.

[4] M. K. Aguilera, C. Karamanolis, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, 2007.

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

[6] C. Bell and D. Bonachea. A New DMA Registration Strategy for Pinning-Based High Performance Networks. In *CAC*, 2003.

[7] 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. http://www.myricom.com.

[8] J. Carter, A. Ranganathan, and S. Susarla. Khazana: An Infrastructure for Building Distributed Services. In *ICDCS*, May.

[9] Intel Corporation. http://www.vnunet.com/vnunet/news/2165072/intel-unveils-tera-scale, Sep 2006.

[10] Jochen Liedtke, Volkmar Uhlig, Kevin Elphinstone, Trent Jaeger, and Yoonho Park. How to schedule unlimited memory pinning of untrusted processes or provisional ideas about service-neutrality. In *Workshop on Hot Topics in Operating Systems*, 1999.

[11] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance vmm-bypass i/o in virtual machines. In *USENIX Annual Technical Conference*, 2006.

[12] Network-Based Computing Laboratory. MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/.

[13] J. Nieplocha and J. Ju. ARMCI: A Portable Aggregate Remote Memory Copy Interface. In *IPDPS*, 1999.

[14] R. Numrich and J. Reid. Co-array fortran for parallel programming, 1998.

[15] Joon Suan Ong. Network Virtual Memory. PhD. Thesis, The University of British Columbia, 2003.

[16] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *Hot Interconnects*, 2001.

[17] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. In *IEEE Computer*, Nov 2004.

[18] M. Schlansker, N. Chitlur, E. Oertli, P. M. Stillwell, L. Rankin, D. Bradford, R. J. Carter, J. Mudigonda, N. Binkert, and N. P. Jouppi. High-performance ethernet-based communications for future multi-core processors. In *Super Computing*, November 2007.

[19] H. V. Shah, D. B. Minturn, A. Foong, G. L. McAlpine, R. S. Madukkarumukumana, and G. J. Regnier. CSP: A Novel System Architecture for Scalable Internet and Communication Services. In *the Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, 2001.

[20] K. Vaidyanathan, P. Balaji, S. Narravula, and H. W. Jinand D. K. Panda. Designing Efficient Systems Services and Primitives for Next-Generation Data-Centers. In *NSF Next Generation Software(NGS) Program;*, 2007.

[21] K. Vaidyanathan, S. Narravula, and D. K. Panda. DDSS: A Low-Overhead Distributed Data Sharing Substrate for Cluster-Based Data-Centers over Modern Interconnects. In *HiPC*, 2006.

[22] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *the 2003 International Conference on Parallel Processing (ICPP 03)*, Oct. 2003.

[23] Pete Wyckoff. Memory Registration Caching Correctness. In *CCGrid*, 2005.