Optimized Page Fault Handling During RDMA

Antonis Psistakis, Nikos Chrysos, Fabien Chaix, Marios Asiminakis, Michalis Gianioudis, Pantelis Xirouchakis, Vassilis Papaefstathiou, and Manolis Katevenis

Abstract—Remote Direct Memory Access (RDMA) is widely used in High-Performance Computing (HPC) while making inroads in datacenters and accelerators. State-of-the-art RDMA engines typically do not endure page faults, therefore users are forced to pin their buffers, which complicates the programming model, limits the memory utilization, and moves the pressure to the Network Interface Cards (NICs). In this article we introduce a mechanism for handling dynamic page faults during RDMA, named PART, suitable for emerging processors that also integrate the Network Interface. PART leverages the IOMMU already present in modern processors for translations. PART avoids the pinning overheads, allows any buffer to be used for communication, and enables overlapping page fault handling with serving subsequent RDMA transfers. We implement and optimize PART for a cluster of ARMv8 cores with tightly-coupled network interfaces. Handling a minor page-fault of a small transfer at the destination takes approximately 38 μsecs, while there is no performance degradation when running three full MPI applications in 16 nodes and 64 cores. Detailed breakdown uncovers the hardware and system software components of this overhead and was used to further optimize the system. A 4MB RDMA transfer performs 1.46x better over pinning.

Index Terms—Page fault, RDMA, IOMMU, MPI, low-power ARM processors, pinning avoidance

1 Introduction

Modern computing systems strive to eliminate the use of the kernel path in processor communication [1], [2], [3], [4], [5]. Kernel involvement induces high overheads due to costly system calls and unwanted memory copies during a transfer. As an alternative, user-level initiation of Remote Direct Memory Access (RDMA) completely eliminates these overheads, by implementing a transport in hardware, and allowing users to bypass the operating system. Originally a High-Performance Computing (HPC) technology, the RDMA is widely used today in datacenters by a variety of applications ranging from web-search [6], key-value stores [7], neural networks [8] and accelerators.

User-level RDMA mandates the use of virtual addresses when specifying the source and destination memory locations. These virtual addresses must be safely translated to physical addresses by the RDMA subsystem in order to access the memory of the communicating processes. Common state-of-the-art RDMA technologies register the address mappings of pages that participate in communication in special Memory Translation Tables (MTTs) in host memory and copy the mappings into Network Interface Cards (NICs), which become responsible for address translation (see Fig. 1). To keep the mappings of registered pages consistent with changes happening in the page table maintained by the OS, *pages are pinned so that their physical locations (frames) do not change while the network accesses them* [9], as shown in Fig. 1. Pinning is undesirable for the following reasons:

1) Extensive use of pinning can hinder the memory utilization [10] and is not compatible with some optimizations of the OS (e.g., Transparent Huge Pages).
2) The applications are responsible to pin and unpin their working set of communication buffers, rendering programming more difficult.
3) Pinning and unpinning pages requires system calls that introduce overheads.

Earlier studies have measured the cost of pinning and registering pages. Their results show that the overhead for a single page can range from a few up to several tens of microseconds, depending on the platform, when, today, an RDMA transfer itself may complete in less than a microsecond [11]. Applications that use RDMA for improved performance wish to hide this start latency as much as possible. In addition, modern NICs deploy large MTT caches in order to serve many concurrent communication buffers. This increases their cost and area footprint, while not necessarily making them capable to capture the working set of the most demanding workloads [3], [12]. The aforementioned state-of-the-art approach hinders the resource management, scalability and latency benefits of "Demand Paging", because communication pages have to always be resident in memory prior to actual access. Furthermore, these solutions typically maintain a secondary memory translation subsystem (in MTTs and NIC cache) for communication buffers, which increases cost and complexity.

- *Antonis Psistakis is with the Department of Computer Science, University of Illinois at Urbana-Champaign (UIUC), Urbana, IL 61801-2302 USA. E-mail: psistaki@illinois.edu.*
- *Nikos Chrysos, Fabien Chaix, Marios Asiminakis, Michalis Gianioudis, Pantelis Xirouchakis, Vassilis Papaefstathiou, and Manolis Katevenis are with the Institute of Computer Science, Foundation for Research and Technology-Hellas (ICS-FORTH), 700 13 Heraklion, Crete, Greece. E-mail: {nchrysos, chaix, marios4, yanoudis, pxirouch, papaef, kateveni}@ics.forth.gr.*

Manuscript received 14 May 2021; revised 29 April 2022; accepted 11 May 2022. Date of publication 20 May 2022; date of current version 23 August 2022. This work was supported in part by European Commission through the Horizon 2020 Framework Programme under Grant Agreements 671553 and 754337, and in part by European High-Performance Computing Joint Undertaking (JU), and by France, Greece, Germany, Spain, Italy, and Switzerland under Grant Agreement 955776 through RED-SEA Project. The JU was supported by the European Union's Horizon 2020 Research and Innovation Programme.
(Corresponding author: Antonis Psistakis.)
Recommended for acceptance by R.M. Badia.
Digital Object Identifier no. 10.1109/TPDS.2022.3175666

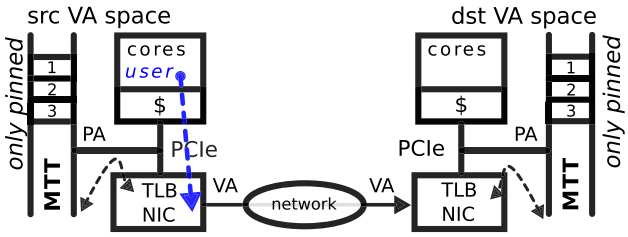


Fig. 1. Pinning in traditional network interface cards (NICs). Pages are pinned prior to the transfer, and the mappings are copied in separate memory subsystem that spans the NIC and main memory.

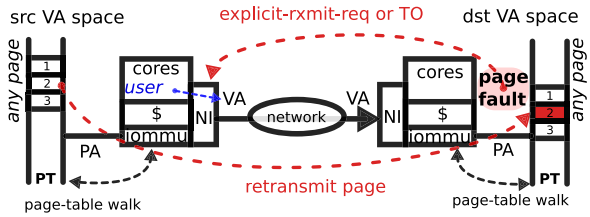


Fig. 2. PART 's general architecture. In PART, we do not pin pages, and memory translations go through the IOMMU, and the page tables (PT) of the user processes. Page faults are handled by re-transmitting failed pages after a timeout, or explicit retransmit messages.

Several works use pre-pinned communication buffers [13], [14]. In this approach data are copied to these communication buffers prior to RDMA. Copying large amounts of data into dedicated buffers incurs overhead for the CPU, may pollute the cache, and also requires additional programming effort.

Our goal in this paper is to enable scalable RDMA without the overheads incurred by memory page pinning, which is achieved by supporting dynamic page faults. The main observation is that in modern well-designed systems, page swapping, and thus *page faults, will be rare, regardless of pinning the communication buffers or not*. This holds especially true for optimized HPC applications that keep their working set in main memory.

Following these insights, we propose and evaluate PART, a system that does not pin the communication buffers. PART leverages the RDMA transport layer, treating occasional page faults similarly with other transmission errors that may occur in a transfer. In a nutshell, PART first resolves the page fault locally at the node that it occurs and subsequently retransmits the failing pages of the RDMA transfer. PART targets applications that their working set does not fit in memory, legacy applications that would require prohibitive complexity to support RDMA by adding pin/unpin code before/after the buffers are RDMAed, and mainly programs with infrequent page faults (for performance).

PART does not pin pages before using them for RDMA; effectively, users can *use any virtual address of their processes for communication*. However, because buffers are not pinned, a network access may generate a page fault; when this happens, a kernel module in PART is invoked, which resolves the page fault, and re-uses the RDMA transport to replay the failing segment of the transfer, as shown in Fig. 2.

PART re-uses the IOMMU that exists in modern servers for network address translation purposes, instead of deploying a separate memory-management system that spans the Network Interface (NI) and the Operating System (OS), as is the case for current NICs shown in Fig. 1. This makes PART an attractive technology for next-generation computing systems featuring a network interface that is tightly-coupled with host and memory [15], [16].

Overall, the memory management proposed in PART is simpler, as user programs do not have to explicitly manage a scarce set of communication buffers; instead, any region in the program's virtual memory can be used for remote memory operations. In addition, the hardware of the network interface is much more efficient, as we re-purpose the IOMMU, which already exists in modern systems, instead of spending resources on an extra memory management subsystem, as is the case with modern NICs.

Our contributions in this paper are the following:

- We propose PART, a system that handles the page faults during RDMA thus removing the need to pin buffers in memory.
- PART re-uses the IOMMU for address translation, instead of relying on specialized NICs, and leverages the RDMA resiliency capabilities to re-transmit failed pages.
- We provide a detailed breakdown of PART 's latency in resolving network-induced page faults. In our ARMv8 testbed, the latency of a small (single-page) transfer incurring a network page fault is approximately 38 μsecs, as detailed in Section 6. This includes (and is dominated by) the time needed at the host to wake up from an asynchronous page fault and to resolve it.
- We propose two optimizations, 1) proactive paging and 2) overlapping paging with replaying transfers. This overlap provides up to 1.93x speedup compared to no overlap (Section 6.1.2).
- We implement both the system software and hardware components of PART for ARMv8 processor, utilizing ARM's SMMU for address translation.
- We evaluate the performance of the system and its bottlenecks running three HPC applications in a cluster of interconnected ARM processors, as well as microbenchmarks, in which we vary the frequency of page faults and discuss trade-offs and possible optimizations. Our results show that PART performs virtually as well as a system that avoids network page faults.

The remainder of this paper is organized as follows. First, in Section 2, we overview the related work. In Section 3 we describe PART, our page fault handling mechanism. Section 4 presents the implementation of PART in an ARM-based platform, which exploits and re-uses ARM's IOMMU (SMMU). Then, in Section 5 we present a thorough latency breakdown of PART, and introduce a number of optimizations. In Section 6, we use extensive microbenchmarks and three real-HPC applications to evaluate PART and to come up with a number of insightful take-aways. Finally, we conclude this work in Section 7.

2 Related Work

RDMA promises to evade software processing overheads during communication. However, because they cannot handle page faults on demand, in hardware, common RDMA

technologies mandate that communication buffers are registered and pinned in memory [9], [17], [18].

An early study measured the pin-down overhead for Myrinet networks as $(40 + 7 \cdot n)$ μsecs, where n is the number of pages [19]. The startup cost of 40 μsecs is paid for the system call and context switch, and 7μsecs is the incremental cost per page. Later, in [20], the total cost of registering a page was found to be approximately 30 μsecs. This cost stays constant for buffers up to 128KB, and is increased proportionally with the buffer size afterwards. For comparison, in our newer ARM-based platform, we measured the latency of pinning a page at approximately 6 μsecs. Out of these 6 μsecs, we found the startup cost of pinning a buffer that is not in memory to be approx. 3 μsecs.

In the past, special memory management techniques for communication buffers have been proposed in order to hide the latency of page registration and pinning. There is considerable work to hide the corresponding overheads using a shared pool of pre-pinned pages [9] [10]. Using pre-pinned communication buffers (data are copied when needed) [13], [14] increases the latency, the CPU overhead, the cache thrashing overheads as well as the programming effort. Lazy de-registration in [19] defers releasing pages anticipating future re-use. In [9], the authors propose to copy small messages in pre-pinned buffers, whereas other studies propose to register the entire physical address space [21]. Compared to these approaches, PART takes a completely different approach, as it does not require the communication buffers to be pinned, thus PART (i) does not limit the size of communication buffers, (ii) does not require special runtimes to manage or share communication buffers, and (iii) performs equally well or even better than previous approaches.

Modern NICs try to help developers to hide the registration latency by offering larger translation caches. The authors in [22] report that the MTT cache in the NIC of their study has at most 2K entries. For 4KB pages, this cache can keep translations for 8MB communication buffers. Cache misses are served by the host MTT that maintains the translations of all pages. This operation takes time, since it goes through the PCIe (approx. 500 nsecs one-way latency in unloaded systems [23]). In high-speed networks, this dragging path can induce a "slow receiver syndrome", which can cause and spread congestion throughout the network. Large page sizes and modern NIC optimizations for contiguous physical memory may help even further in these directions [24]. Note that congestion spreading due to SMMU TLB misses can also incur in PART , possibly not that extensive because the path to memory is faster (no PCIe). A page table walk involves accessing all the levels of the page table, and in our system this overhead is approximately 600 ns.

A previous work, independent of ours, resolves network page faults dynamically [25]. In a similar approach with us, buffers are paged-in when the Host Channel Adapter (HCA) needs them and pages them out when the OS requests them. Therefore, there is no need to pin pages. As noted in the paper, modern Mellanox InfiniBand NICs support many of these technologies, such as on-demand paging (ODP) [26], [27]; recent advances in current InfiniBand verbs allow registering the entire address space for ODP [28]. Next, we compare this scheme against ours.

- PART re-uses the IOMMU in order to have direct access to processes' page tables. In contrast, [25] needs to explicitly bring-in and -out mappings, as mandated by either the application or the OS.
- PART introduces and leverages the overlap of serving a page fault with transferring data in order to achieve better performance.
- In this paper we implement proactive paging and demonstrate its performance benefits, while [25] only mentions it as a possible optimization.
- PART is implemented on a different platform than [25] (ARM with FPGA-based NIC, versus x86 with Mellanox-based NIC).
- The overhead of the mechanism in [25] for 4KB messages is approximately 220 μsecs, due to the slow NIC firmware. For comparison, albeit measurements are on different platforms, PART resolves same-sized page faults in approximately 38 μsecs.

Reading the related literature, it seems there is no one-solution-fits-all when it comes to when pinning should take place: it depends on the application and the implementation as well. However, pinning per transfer is a non-viable solution due to system call overheads. On the other hand, one-time pinning, e.g., pinning a potentially large address space, poses several performance challenges, such as hurting the memory utilization. All these problems are completely avoided when using PART. In [29], the authors find that handling page faults dynamically, as we do in this paper, but using ODP [28], is very expensive. However, our mechanism is one order of magnitude faster than the one in [29].

Although in our work we have enabled seamless page table sharing between the CPU and the RDMA Engine through ARM's SMMU, there are several similar technologies which achieve the same goal. SVA (Share Virtual Address) allows devices to have access to memory using the same virtual address with the CPU. Intel-SVM (Share Virtual Memory) allows CPU's MMU and a device's IOMMU to share the page table [30]. Nvidia-UVA (Unify Virtual Address) uses different page tables for GPU and CPU. GPU-MMU mirrors CPU page table and allows accessing the same data by having a copy of CPU's DDR to GPU's HBM. ARM's SVA (Share Virtual Address Space) uses a SMMU instead of an IOMMU [31] –with the help of the cache coherent interconnect (CCIX), coherence among different devices and CPU can be achieved. Our scheme complemented the ARM SMMU we used, because by itself it did not support SVA.

3 PART : Handling RDMA Page Faults

When issuing RDMA operations, the users specify the source and the destination node IDs as well as the source and the destination virtual addresses. Virtual addresses are used in RDMA for virtualization and protection purposes. Effectively, when the communication buffers are not pinned, a page fault might occur in:

- the source buffer
- the destination buffer
- both buffers

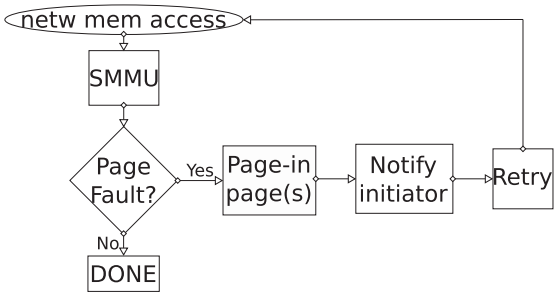


Fig. 3. PART mechanism general flow.

In this section, we present how PART treats minor page faults, either at source or destination. Minor page faults occur when the page table does not contain a mapping to a physical page, but can be resolved without involving I/O (e.g., disk), in contrast to major page faults. They typically occur when accessing a newly allocated buffer (due to kernel's Demand Paging [32]), or shared data that belong to a different process.

The most common case for a page fault to occur during an RDMA is at the destination buffer, because the source buffer (containing the transfer data) will typically be written ("touched") prior to the transfer. On the other hand, a user process may allocate a (receive) buffer which is uninitialized when the RDMA starts writing data into it. Effectively, the access will result in a *minor page fault* at the destination buffer. Major page faults are outside of the scope of this work; nonetheless, PART is expected to handle major page faults during RDMA in a similar manner with minor page faults.

Page faults at source can occur, among other reasons, due to internal memory optimizations such as Transparent Huge Pages (THP), which is enabled by default in some Linux kernel distributions [33]. THP works quietly in the background, substituting physically contiguous pages for huge pages into a process' address space [34], enabling all benefits of huge pages such as less TLB misses.

Fig. 3 depicts the general flow of PART resolving a network page fault. An RDMA memory access to the destination computing node reaches first the RDMA Engine, then passes through the IOMMU (SMMU), and finally, if allowed permission-wise, accesses the memory. A page fault will occur when the targeted page is not resident in memory. In this case, the IOMMU will invoke a fault handler.

The first task of handling page faults is to make sure that the failed page is brought into the main memory, as long as it is valid. In PART , we accomplish that in kernel-space, utilizing the `get_user_pages()` and `put_page()` methods of the kernel.[1] The former essentially pins-in a range of pages, thus bringing them also in memory, and the latter unpins them.

3.1 Proactive Page-In

Once we encounter a page fault on an RDMA transfer, PART can *proactively page-in more than one page* of the transfer. In Section 6, we evaluate the performance of *Page-in 1-Pg*, which pages in only the failed page, *Page-in 4-Pgs*, which proactively pages-in four pages (i.e., 16 KB) and, finally, *Page-in All-Pgs* which pages-in all pages of an RDMA transfer, starting from the failed one. If the size of the transfer is unknown, the get_user_pages() kernel method can bring in unnecessary pages belonging to the process, outside of the transfer scope. This may induce side effects, such as cache trashing, but does not affect correctness.

3.2 Timely Page Retransmission

PART relies on the RDMA transport in order to replay faulty pages. When a page fault occurs, the source RDMA Engine will not receive a positive acknowledgment (ACK), and will retransmit (in hardware) the failed block after a time-out. The timeout of an RDMA Engine cannot be set to be a meager value, because this can induce early timeouts and duplicates. Thus, in a network with a few microseconds end-to-end latency, the timeout may be set at 100 μsecs or more.

Retransmissions caused by failing packet Cyclic Redundancy Check (CRC) do not have to wait for a long timeout, but can be expedited using negative acknowledgments (NACKs). PART does not rely on such NACKs generated when a page fault occurs, because the re-transmission may then arrive before we have paged-in the missing page(s). In order to expedite the re-transmission, without relying on the timeout value, PART sends an *explicit re-transmission request (ERR)*, after the pages are in memory, as shown in Fig. 2.

Note that if proactive page-in is enabled, the source may resume a transfer (sending the failed and new segments) after its timeout expires, while subsequent pages are still paged-in. We discuss this overlap between transfer and page-in in the following subsection.

3.3 Overlapping Paging-In With Data Transfer

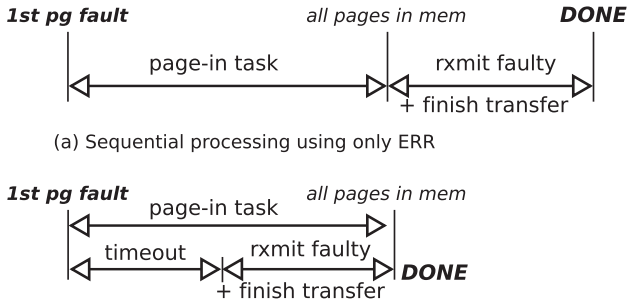
Network page faults induce overheads to RDMA, both in terms of host intervention and retransmissions. Page faults are tolerable as long as they occur rarely. However, in PART these overheads do not add up, since they can overlap.

PART can service subsequent transactions (such as the first replayed data), in parallel with handling additional page faults from the same or different transfers. To enable this, we use a Hit Under Previous Context Fault (HUPCF) configuration of ARM's SMMU, discussed in Section 4.3.

While the page faults are resolved, PART does not send an ERR message. However, the source may timeout and resume the transfer independently, starting from the missing pages, and continuing with new ones, which have also been brought into memory by PART. Effectively, by the time the source node sends the data of a page, the corresponding pages will be in-memory even though they were not initially there at the beginning of the transfer. In this case, the transfer time can overlap with page-in time.

Let $T(n)$ denote the transfer latency of n number of pages, $G(n)$ the time of the PART to page-in n pages, and TO the retransmission timeout period. Then the speedup gained by the overlap, $s(n)$, can be approximated by the following formula:

1. Another way to accomplish that is to have a user-space thread, linked with the user-process executable, be informed about the fault, so that it touches (i.e., reading and then writing the first byte of) the failed page; we preferred the kernel-based solution, because it avoids utmost context switches, and also because it enables some optimizations, that will be described later in the paper.



(a) Sequential processing using only ERR

(b) Overlap between page-in & transfer using time-out

Fig. 4. Time flow of (a) sequential execution of page-in and (b) ideal execution, in which (proactive) page-in, and transfer, including the retransmission of faulty page, overlap.

$$s(n) = \frac{G(n) + T(n)}{max(G(n), TO + T(n))}. \quad (1)$$

We schematically demonstrate the overlap between transfer and page fault handling in Fig. 4. The figure depicts an ideal case, wherein the retransmit overlaps completely with the page-in task. In reality, the timing may not be exact: either the retransmit or the process of paging can lag behind; however, this does not compromise correctness.

For small transfers, ERR allows PART to timely retransmit the failed page after the page fault has been resolved, without having to wait for a typically longer timeout –see Section 5.2. For larger transfers, however, the ERR alone, i.e., sending it after resolving all page faults, does not permit the overlap. Thus, for larger transfer, the no-time-out solution (i.e., ERR alone) performs worse, as we discuss in Section 6.1.2.

We note that one could send the ERR before resolving all page faults (or even before starting handling them in some cases, e.g., for small transfers) in order to achieve a better overlap without relying on the timeout. Profiling of when to send the ERR based on the number of pages of a transfer or the workload could be part of future work.

4 PART for ARM-Based Platforms

In this section, we present our implementation of PART in an ARM-based platform. In our evaluation sections (Sections 5 and 6), we collect results using this implementation.

4.1 Testbed

The basic block of our testbed is the Quad-FPGA Daughter Board (QFDB) [35], [36], providing 4 Zynq MPSoCs (F0-F3) [37], interconnected through point-to-point 17 Gb/s links, as shown in Fig. 5. Each QFDB additionally provides 10 10Gb/s links that are available through F0 for connections with the outside world. Every MPSoC provides four ARMv8 cores (A53), running at 1.2 GHz, additional Xilinx IPs –IOMMU, L2 caches, Cache Coherent Interconnect (CCI)– as well as a programmable logic (FPGA) part. Every MPSoC also has access to 16GByte of private DDR4 memory. The cores are fully coherent with each other, whereas the programmable logic can access memory through IO-coherent ports. Our testbed consists of several QFDBs connected in a Torus topology[35], [36]. The latency of each hop in our prototype is approximately 150 nsecs, and the base (min) latency of RDMA operations is around 4 μsecs.

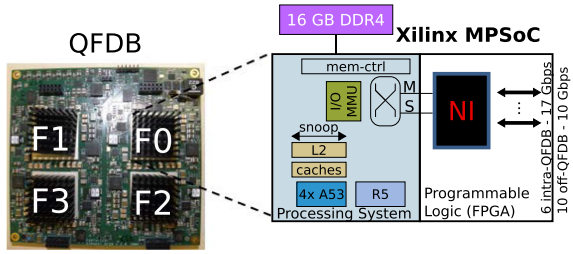


Fig. 5. PART's building blocks: (left) Quad-FPGA Daughter Board (QFDB) with 4 Zynq MPSoCs; (right) Each MPSoC provides 4 ARMv8 cores, 16GB DDR4 (connected with a parallel bus at 160 Gb/s), an IOMMU, and a custom RDMA transport (NI) implemented in the FPGA.

4.2 Reliable RDMA Transport Utilizing ARM's SMMU

A custom *network interface (NI)* inside the FPGA part of the MPSoC implements a packet-based network protocol for inter-chip communication. Processes initiate RDMA (write and read) transfers using virtual addresses, bypassing completely the OS. The NI is highly virtualized, offering multiple channels that can be allocated to concurrent user-level processes or threads. The NI is connected with the CCI and the main memory via an Advanced eXtensible Interface (AXI) interconnect. In our measurements, the round-trip time between an ARM processor and the NI is between 120 and 150 nsecs, which corresponds to the latency of a processor load command that reads a register inside the Programmable Logic.

Address Translation Using SMMU. Instead of using a special subsystem, i.e., Memory Translation Table (MTT) buffers at the host and translation buffers at the NIC, PART makes use of the System Memory Management Unit (SMMU) [37]. The SMMU is defined by ARM, but, operation-wise, is similar to other IOMMUs. Traditionally, the SMMU is responsible to manage the memory requests from I/O devices to the local memory of the system, but here we re-purpose it for accesses coming from the network. Note that a page fault requires invoking the kernel in order to be handled, while a SMMU TLB miss can be handled by the hardware Page Table Walker without any software (or OS) intervention if the page is in memory.

When issuing RDMA operations, the users specify the source and the destination node IDs (22-bits each) as well as the source and the destination VAs, 42-bits each. In order to identify the targeted process at end-points, and provide protection, we deploy a special 16-bit *protection domain identifier (PDID)* on the RDMA channels and carried by network packets, which are allocated to the processes by the OS.

The NI uses these {PDID, VA} tuples to access the memory of user-level processes. The PDID is used to uniquely match incoming memory transactions to a particular structure of the SMMU, called *context bank*. Each context bank is associated with the page table of a process.

Accesses to host memory coming from the NI pass through the ARM's SMMU, which translates virtual to physical addresses, as described in more detail in the next subsection. Next, the physical addresses are forwarded to ARM's Cache Coherent Interconnect (CCI) to fetch (on read) or invalidate (on write) cached data inside the Level-1 and Level-2 caches. Effectively, we do not need to flush the caches before triggering RDMA operations.

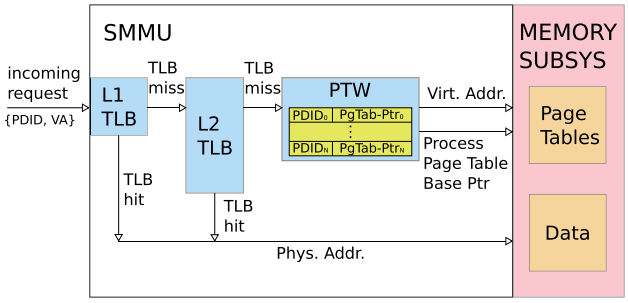


Fig. 6. Memory translations through ARM's SMMU subsystem.

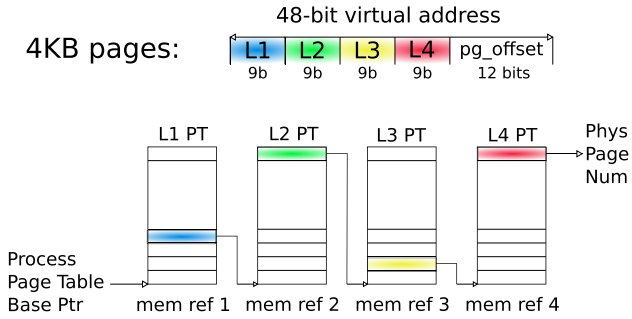


Fig. 7. A page table walk for a 48-bit address involves four different memory accesses before reaching the Physical Page Number (PPN).

Reliable Transport. The NI offers a reliable transport, based on end-to-end, block-level positive or negative acknowledgments and retransmissions, allowing transfers to complete in the presence of errors without any kernel or software overhead. Unless otherwise noted, the block-level retransmission timeout value is 1ms, but, in our evaluations, we experiment with other values as well.

Part of the RDMA Engine is the R5 Real-Time co-processor available in the Zynq MPSoC. Its main task is to segment messages into 16KB blocks (or transactions) and to issue block transfers to the hardware engines. The hardware engine further splits 16 KB transactions into 256 Byte packets. At the source, it reads packets' payload from host memory starting from the source virtual address (VA). At the destination, the RDMA Engine writes the packets' payload to host memory, through the SMMU, starting from the destination VA.

For each correctly completed block, a positive ACK is generated, which is routed to the R5 processor at the source. The engine can also send a NACK, with special qualifiers indicating whether a received packet was corrupted or if the memory subsystem (e.g., the SMMU) returned a NACK when accessing memory (e.g., because the page is not in memory). Missing ACKs trigger timeouts and block-level retransmissions. NACKs can also trigger retransmission, depending on their qualifier. PART does not retransmit a block upon receiving a page fault NACK generated by the SMMU; instead, the source waits for the ERR, which is sent after the faulty pages (with proactive paging) are in memory, or for the timeout in order to retransmit.

The hardware end-to-end latency (the latency of the network including the RDMA engines that fetch data from memory at the source and write data to destination memory) is below 1 μsec; the R5 co-processor adds approximately 3 μsecs for the initiation of the transfer.

End-to-End Flow Control. The source limits the number of outstanding transactions for each transfer (2 blocks in our implementation). Therefore, if an ACK is missing or a NACK is received, the transfer can be suspended, until the ACK is received (possibly after the block has been re transmitted).

4.3 ARM's System Memory Management Unit

In this section, we present the internal operation of the SMMU used, outlining its translation cache capabilities and the page-walker. The SMMU includes Translation Lookaside Buffers (TLBs) that keep the most recent address translations, without needing to perform a page table walk (PTW). The SMMU embeds *two levels of TLBs*. The Level-1 TLB of the SMMU used in this work is a fully associative cache and can support up to 128 entries, while the Level-2 TLB is a 4-way associative cache and can support up to 2K entries, thus offering translation capabilities comparable with that of modern NICs. The SMMU in our platform supports 256 outstanding transactions and up to 16 parallel PTWs for every Level-2 TLB.

One difficulty in using a particular SMMU version for network addresses is that the TLBs of the SMMU are not coherent with the page table of the OS. For this reason, we invalidate the SMMU's TLB entries when the OS modifies existing entries in the page table. The invalidation of the SMMU TLB with an integrated SoC is likely to be comparable to invalidating a CPU core TLB. For applications that fit in memory and reuse buffers, changes in the page table are expected to be infrequent.

Due to this, in an adversarial scenario, the SMMU may not find mappings in its caches because of unrelated memory management activities. However, these misses will be resolved in hardware by the Page Table Walker, with up to 4 accesses to memory, without needing the help from the application or from the operating system. Note that this is a limitation of the current SMMU implementation in our testbed and not an issue related to PART itself. A more fine-grained approach is described in the specification of the SMMU [38], which can be used for TLB invalidations.

In Fig. 6 we see the translation of an incoming transaction ({PDID, VA} tuple) passing through the SMMU. When a mapping is not found in the TLBs, a *PTW* is triggered on the process page table, which is maintained in DRAM. A PTW performs a number of memory accesses that can degrade the performance of a system.

In Fig. 7, we see an example of a PTW for a 48-bit virtual address. As can be seen, the PTW completes after four memory accesses to the main memory. However, the PTW of the SMMU incorporates extra caches that can speedup the translation process by taking advantage of spatial and temporal locality in memory accesses. For instance, the SMMU maintains a cache of the contents of the Level-3 Page Table (PT) as well as a pre-fetch buffer, reducing the number of memory accesses. Note that our CPU effectively utilizes 39-bit virtual addresses, and thus only three levels of translation are involved.

Hit Under Miss. ARM's SMMU has a mode called Hit Under Previous Context Fault (HUPCF), which enables the device to process all upcoming transactions regardless of other possible outstanding context faults (e.g., a page fault).

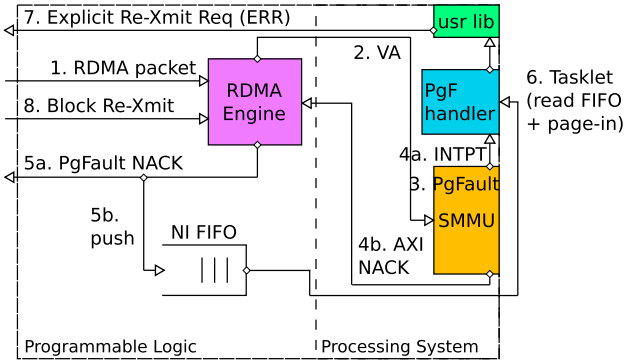


Fig. 8. Handling an RDMA page fault at the destination when an RDMA packet arrives.

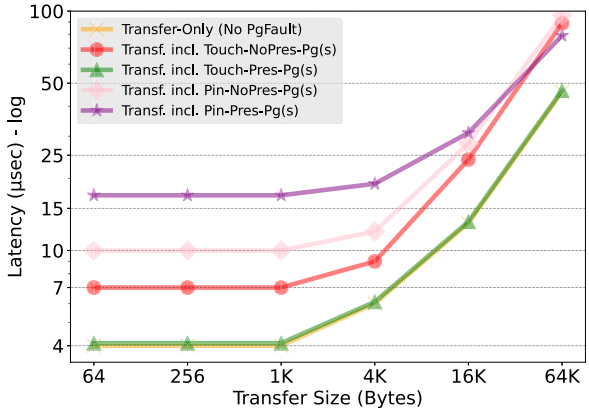


Fig. 9. Latency of RDMA (no page fault); Two curves are on top of each other: *Transfer-Only* and *Transf. incl. Touch-Pres-Pg(s)*.

4.4 Resolving a Page Fault in ARM-Based Platform

Fig. 8 depicts in more detail how PART handles a page fault. The RDMA Engine tries to read or write data to local memory using ARM's AXI memory interconnect. The corresponding AXI write request has an Input/Output (IO) virtual address which is translated by the SMMU before accessing the local memory. As can be seen in Fig. 8, the request generates a page fault, which triggers an interrupt that is captured by the SMMU fault handler. In parallel, an AXI NACK arrives at the local RDMA Engine, which logs in a Network Interface FIFO Queue (NI FIFO) all the necessary information needed in order to resolve the page fault and to request from the sender later to replay the failed block.

In parallel, the page fault handler schedules a tasklet that will resolve the page fault. In general, tasklets are preemptable and allow the interrupt handlers to be released sooner, reducing the response time for other interrupts. The tasklet starts by reading all entries in the NI FIFO, and identifies unique pages that experienced page faults—a single page may correspond to multiple failed AXI-write requests, and thus to multiple AXI-NACKs registered in the NI FIFO. For each unique failed page, the tasklet resolves the page fault (s), using `get_user_pages()` and `put_page()`, as discussed previously in Section 3, and then triggers an ERR to the sender.

In order to send the ERR to the initiator node, in our implementation we utilize Netlink sockets [39] to communicate the corresponding information from kernel space to a userspace library. The userspace program then issues an ERR message to the R5 co-processor of the source node, in order to resume the transfer, by replaying first the faulty pages.

In our implementation of PART, we selectively retransmit only the block of the failed page. For networks that *do not feature selective retransmissions*, it may be beneficial to proactively page-in all-pages upon the first page fault (or even pre-touch the buffers, as discussed in Sections 3.1 and 6), in order to avoid multiple retransmissions of the entire transfer, which can occur when multiple pages fail. We should note that our FPGA-based RDMA supports selective retransmission of failed blocks, because of its rich capabilities in bookkeeping outstanding blocks of 16KB and their ordering. More specifically, it utilizes a transaction identifier (id), which uniquely identifies each transmitted 16KB block of a transfer, and a sequence number, that upon a retransmission is incremented; therefore, older packets (i.e., with smaller ids) that arrive later in time will be dropped.

5 PART : Breakdown & Optimizations

In this section we evaluate PART using custom-made synthetic microbenchmarks. Unless noted otherwise, every experiment is repeated at least 5 times; some experiments, with large variance in the measurements were repeated many more times. We report the average of the individual measurements.

5.1 Ideal Execution of an RDMA Transfer

In Fig. 9, we depict the latency of RDMA writes for different message sizes. In this experiment, *no page fault occurs when we access memory from the network*. With *Transfer-Only*, we measure the user-perceived latency of PART, when all pages are in memory.

For comparison, we also depict the user-perceived latency when pages are pinned before the transfer, using the `mlock` system call. We present two different scenarios for pinning; one that includes the overhead of calling `mlock()` for a buffer that pages are already in memory (*Pin-Pres-Pg(s)*), and another that includes the overhead of calling `mlock()` for a buffer that pages are not in memory (*Pin-NoPres-Pg(s)*); in the latter case, pages are accessed for the very first time when calling `mlock()`. Surprisingly, we notice that when pages are already in memory the overhead of pinning a buffer less than 16KB in size is higher compared to when pages are not in memory. We suspect this overhead is from the OS that may have to perform additional operations such as copying and moving data, but we leave further investigation on this to future work. In the case of *Pin-NoPres-Pg(s)*, pinning adds 6 μsecs for small ($<$4KB) transfers (10 versus 4 μsecs), while in the case of *Pin-Pres-Pg(s)*, the corresponding overhead adds approximately 13 μsecs to the total transfer time; this overhead increases with the transfer size, i.e., with the number of pages pinned, nearly doubling the latency of the operation.

Fig. 9 also depicts the overhead of *touching* pages instead of pinning them. To account for this, we include in our measurements the latency of the user process or the runtime reading and writing the first byte from each page prior to transfer. As with the case of pinning, pages can either *be*

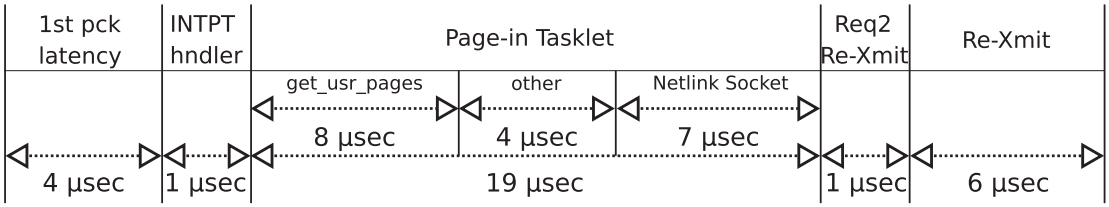


Fig. 10. Breakdown of a 4KB RDMA transfer that exhibits a page fault (resolved by PART).

not-present in memory, e.g., first access to a buffer (*Touch-NoPres*), or they might already reside in memory (*Touch-Pres*). The user/runtime cannot know the state of each page without system calls. Thus, touching pre-faults the buffers that are not resident in memory, without pinning them, while invoking the kernel only on these (faulty) pages.

Touch-NoPres, similarly to *Pin-NoPres-Pg(s)*, exhibits an internal minor page fault on each page. As can be seen, for small transfers the overhead of *Touch-NoPres* is less than that of *Pin-NoPres-Pg(s)* (3 versus 6 μsecs overhead), but the latency of these methods converge for larger sizes. On the other hand, the overhead of touching a page already in memory is around 100 nsecs, thus *Touch-Pres* nearly overlaps with *Transfer-Only*. However, for larger transfers, the overhead is considerable, reaching approximately 20 μsecs and 152 μsecs for 1MB and 4MB transfers, respectively, in our measurements. Furthermore, we notice that *Pin-Pres-Pg (s)* appears to be more expensive than almost all other cases in all sizes (except 64KB) presented in Fig. 9.

The added latency of touching pages before RDMA transfers becomes increasingly important in *asynchronous* transfers, such as *MPI_Isend*. As discussed further in Section 6.2.1, the touch operation can delay such asynchronous calls and keep a CPU core busy for many 10s of μsecs (in large transfers), which is clearly undesired.

Take-Aways. 1. Pinning and touching pages not present in memory become identical cost-wise when the buffer size increases, because they are dominated by the cost of the MMU page fault. *2.* For small transfers up to 64KB, the cost of touching pages already in memory (green line/triangles) prior to each transfer is negligible, and thus could be used together with PART 's dynamic page fault support, in order to avoid (but not necessarily eliminate) page faults during RDMA transfers. *3.* Pinning buffers repetitively (and unreasonably) prior to RDMA is not efficient because of the system call overhead. This cost will mostly be pronounced if the buffer is in memory (*Pin-Pres-Pg(s)*). Compared to this, a repetitive-touch strategy is better in terms of performance, because there is no system call involved.

5.2 Overhead of Network-Induced Page Faults

Inevitably, a page fault during an RDMA transfer leads to an overhead because it is handled by the operating system that needs to update the page table. In Fig. 10, we present the breakdown of latency of a 4KB RDMA write transfer that experiences a minor page fault at the destination.

- The first 4 μsecs is the latency that spans from the time that the user issues the RDMA operation until the first packet reaches the destination. The TLBs of the SMMU will then experience a miss, and the PTW will also fail, since the translation may not find a valid entry in the page table.[2]
- The page fault then invokes the interrupt handler of the SMMU (1 μsec in our measurements), which schedules the page-in tasklet that will resolve the page fault.
- The tasklet is the most time-consuming component of our breakdown (19 μsecs to resolve one translation fault): it reads the page fault information of all entries from the NI FIFO (up to 16 read commands in our setup) and pages-in the faulty pages.
  The work of the tasklet mainly involves:
  - Calling the method `get_user_pages()`, which consumes approximately 8 μsecs.
  - Sending an explicit retransmission request (ERR) from userspace in our design requires the use of Netlink sockets, which transmits information from kernelspace to userspace. The cost of the Netlink socket part from kernelspace is approximately 7 μsecs.
  - The remaining 4 μsecs out of 19 μsecs corresponding to the page-in tasklet are due to other operations in the tasklet, such as packing the transfer meta-data that are communicated to the userspace process through the Netlink socket.
- After bringing the pages in memory, we send an ERR to the initiator, which adds approximately 1 μsec latency. This is the cost as evaluated in userspace upon the arrival of the message from kernelspace through a Netlink socket.[3]
- Finally, retransmitting the missing page adds 6 μsecs, for a total latency of 31 μsecs.

In Fig. 11, we measure the latency of small transfers that incur a page fault at the destination. We examine two different methods to trigger the retransmission: first using ERRs, named *Fast Re-Xmit* in the figure, and, second, waiting for the source RDMA Engine to time-out, named *TOut* in the figure. For the time-out periods, we chose 100 μsecs and 1 msec. The latter is the typical value we use in our cluster in order to avoid early retransmissions, whereas the former, is a more aggressive setting, which is nearly 25 times higher than the base latency of an RDMA transfer (nearly 4 μsecs). Nevertheless, the 100μsec time-out is approximately 3 times greater than the time needed to resolve a page fault, hence it puts a safeguard against early retransmissions while PART is resolving a page fault. In any case, our retransmission time-out settings are on par with those examined in the literature for RDMA networks, e.g., in [40].

As can be seen in Fig. 11, using ERRs (*Fast Re-Xmit*) the total latency of the 4KB RDMA transfer is approximately

2. The breakdown does not include these operations, since they contribute a few hundreds of nanoseconds to the overall latency.

3. This could also be done from kernelspace –subject to future work.

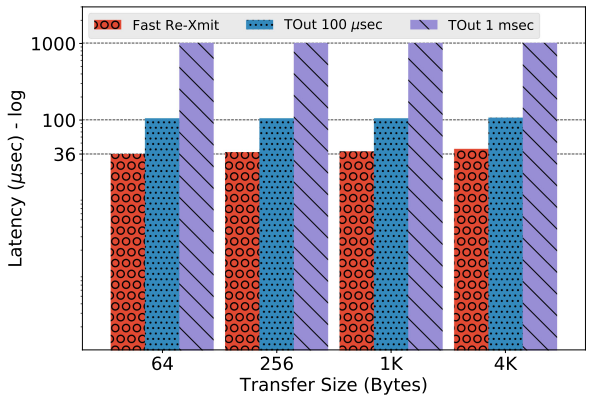


Fig. 11. Latency of RDMA with page fault at destination (resolved by PART) for different retransmission methods.

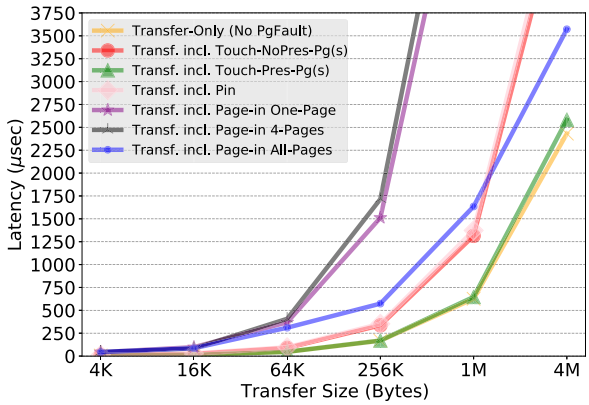


Fig. 12. Latency of an RDMA transfer with page faults in all destination pages being resolved by PART.

38 μsecs, i.e., 7 μsecs higher than what we measured in our breakdown. We believe that this discrepancy is mostly due to OS costs such as context switches, the overhead of which is not included in Fig. 10. In the cases of *TOut*, the latency is dominated by the timeout period. In general, Fig. 11 shows that the latency is similar for transfers up to 4KB, and that ERR is much more efficient than waiting for the time-out.

Take-Aways. 1. For a 4KB page the most time-consuming part of handling a page fault is to page-in the corresponding page. *2.* Systems that do not support ERR can rely on RDMA timeouts, but with some overhead penalty when the paging-in of pages is faster than the timeout period.

5.3 Proactively Paging Ahead Upon a Page Fault

In this section, we examine PART optimizations that pages-in additional pages of a transfer upon a page fault in the first page.

We examine three schemes:

1) *Page-in One-Pg (PIO)* stands for the default scheme, when PART handles every page fault independently.
2) *Page-in 4-Pgs (PI4)* brings in all pages corresponding to the block of a failed page. As mentioned in Section 4, our RDMA Engine splits every transfer into blocks of 16KB (4 pages of 4KB), and selectively retransmits the failing blocks. Therefore, PI4 strives to avoid multiple retransmissions of the same block.
3) *Page-in All-Pgs (PIA)* proactively pages-in all forthcoming pages of a transfer upon the first page fault.

In Table 1, we see all the schemes and their definition, that are used in this section and in the rest of the paper. In Fig. 12, we evaluate a scenario when all pages of a transfer exhibit a page fault, with *Transfer-Only (No PgFault)* being the only exception. In addition to the aforementioned schemes, for comparison we also include some configurations from Fig. 9, with no network page faults, but now the tests extend for up to 4MB transfers –due to lack of space we report only the Pin-NoPres-Pg(s) scenario for mlock(), depicted as *Pin*. As can be seen, PI4 behaves similarly or even better than PIO. With PIO, all pages in the same block will incur a page fault, which the handler of PART can discover and resolve separately, but in one invocation of the handler. Effectively, PIO does not need multiple retransmissions to page-in all pages in a block. Furthermore, PIO finishes its page-in operation (tasklet) and triggers the retransmissions faster than PI4.

PIA, i.e., proactively touching all pages of a transfer, has the same latency for transfers up to 16KB (1 block, 4 pages). However, for larger transfers, the benefits of PIA are enormous. PIA minimizes the number of retransmissions and handler invocations, since, beside the first two outstanding blocks of the transfer, all subsequent blocks will succeed on the first try (pages will reside in memory).

In Fig. 12, the performance of PIA is worse than "Touch NoPres-Pgs" for transfers up to 1MB. More precisely, as can be seen in Fig. 12, for 1MB transfers, PIA is 2.6x worse than "Transfer-Only", whereas "Touch NoPres-Pgs" is 2x worse. Comparing the two, PIA is approximately 1.2x worse than "Touch NoPres-Pgs" for 1MB transfers and 3.5x for 64KB transfers. Although both methods incur the latency of paging-in all pages, through normal CPU MMU minor page faults, "Touch NoPres-Pgs" performs simple load and stores, from user-space prior to the transfer, whereas PIA calls the `get_user_pages()` from inside the paging tasklet, which can be interrupted multiple times due to the SMMU handler invocations. Despite the possible slowdown, as mentioned earlier, PART touches the pages dynamically only when needed,

TABLE 1
Methods Used in the Evaluation of PART

Scheme	Definition
Transfer-Only	No page fault
Touch-NoPres-Pg(s)	Pre-touch all pages (prior to transfer) for the first time (minor CPU page fault)
Touch-Pres-Pg(s)	Pre-touch all pages that are already in memory (no page fault)
PIO (Page-In One-Page)	Upon netw. page fault, page-in only one page
PI4 (Page-In 4-Pages)	Upon netw. page fault, page-in up to block-size number of pages
PIA (Page-In All-Pages)	Upon netw. page fault, page-in all remaining pages of the transfer

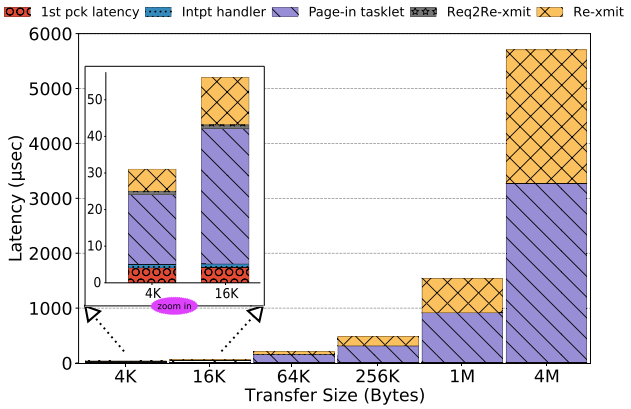


Fig. 13. Latency breakdown of transfers that exhibit page fault(s) at destination (resolved by PART), for various transfer sizes.

whereas "Touch NoPres-Pgs" introduces a static overhead on all transfers, which is especially noticeable for large asynchronous messages, as we discuss in Section 6.2.1.

The situation is reversed for 4MB transfers: PIA is 1.46x *better* than "Touch NoPres-Pgs". This happens thanks to the overlap between PART's paging-in and re-transmission, which we discussed in more detail in Section 3.3.

As a continuation of Fig. 10, Fig. 13 depicts the breakdowns for transfer sizes ranging from 1 page (4KB) to 1024 pages (4MB) when page faults are handled on the fly. We notice that the higher the transfer size, the more dominant the time for the tasklet, which is responsible to page-in the corresponding pages. The cost of `get_user_pages()` that is part of the page-in tasklet, seems to follow a particular pattern: every time it is called, there is a fixed (constant) cost of approximately 6 μsec and a dynamic (incremental) cost based on the number of pages that are not in memory. In our measurements, the dynamic cost is approximately 3 μsec per page brought in memory (upon a minor MMU page fault).

One inconsistency is noticeable in Fig. 13: the expected breakdown for 4MB transfers sums up to 5.7 ms, while in our measurements the total latency is approximately 3.6 ms, i.e., 1.58$\times$ faster. This happens because of the overlap discussed in Section 3.3.

5.4 Pre-Touching Pages of Small Transfers

Our results depicted in Fig. 12 indicate that *it may be beneficial to touch the buffers of small transfers (e.g., less than 64 or 256KB), especially when the user can know that this is the first use of the buffer*.

With pre-touching off the critical path (Transfer-Only (No-Pgf)), the latency is reduced by as much as 3.2x in case page faults occur (Transfer-Only (No-Pgf) versus PIA for 256KB). The latency is marginally increased if pre-faulting of buffers is part of the measurement (Touch-Pres-Pg(s)), because touching a page already in memory occurs an overhead of approximately 100-200 ns.

However, if we know in advance that the buffers will be re-used many times, the cost of pre-touching every time can induce higher overheads than the cost of handling the page fault (only the first time). In the latter case, the unneeded cost of touching the buffers, due to lack of knowledge whether the pages are present in memory or not, can be avoided. This case is also covered and evaluated in Section 6.2.1.

Touching pages is orthogonal to PART , and in certain cases, it can complement PART. When used alone, touching cannot guarantee that no page fault will occur during the transfer. Occasional page faults on accesses coming from the network may still occur if the working set does not fit in memory or when the OS re-allocates pages (e.g., using Transparent Huge Pages).

Take-Aways 1. As shown in Fig. 10, the most time-consuming part of page fault handling is paging-in all the corresponding pages. In our platform, this cost can be estimated by a fixed cost of about 6 μsecs and a dynamic, incremental cost of 3 μsecs per page. *2.* When all pages experience a page fault, PIA outperforms both PIO and PI4, as it will bring in memory all necessary pages before they exhibit a page fault. *3.* The cost of handling page faults dynamically on network accesses can be higher than handling them using page pinning or touching, prior to transfer, from the host. However, PART can allow overlap of paging-in and retransmissions, which leads to better latency than both pinning and "Touch NoPres-Pgs", even when all pages exhibit a page fault.

6 PART Evaluation

Having established the main components in PART 's latency, and introduced a number of optimizations, in this section we continue with additional evaluations, including a sensitivity analysis based on different page fault frequencies and timeouts, as well as the evaluation of real HPC applications, such as LAMMPS [41], HPL [42], and HPCG [43], [44].

6.1 Sensitivity Analysis

6.1.1 Impact of Page Fault Frequency

In the following experiment we evaluate the overhead of handling page faults with PART if only a portion of pages in a transfer exhibits a page fault. We use the following notation: s is the number of pages in the transfer and f the probability that a page exhibits a page fault. Every page of the transfer has the same independent probability f to exhibit a page fault, which we evaluate using POSIX rand(). We note that someone could investigate distributions to generate the pages that exhibit a page fault because e.g., applications may have particular locality attributes. This is something that could be investigated in future work.

Using our notation *page faults occur only for configurations with* $s \times f \geq 1$. For example, for 4KB transfers, i.e., $s = 1$ number of pages, page faults occur only when $f = 100\%$; effectively, when the page fault frequency is up to 80%, the latency is virtually identical to the base latency, with no page fault. For 16KB transfers, the corresponding threshold value in our experiments is $f = 40\%$, for 64KB it is $f = 20\%$, for 256KB $f = 5\%$, and for larger transfers $f = 1\%$.

In Fig. 14, we evaluate the performance of PART (PIO and PIA) versus the page fault frequency for different transfer sizes. With page fault frequency $f = 0\%$, we capture the no page fault case. The left y-axis depicts the latency of the RDMA transfer for various page fault frequency scenarios. The right y-axis presents the slowdown of each configuration compared to the case in which no page fault occurs

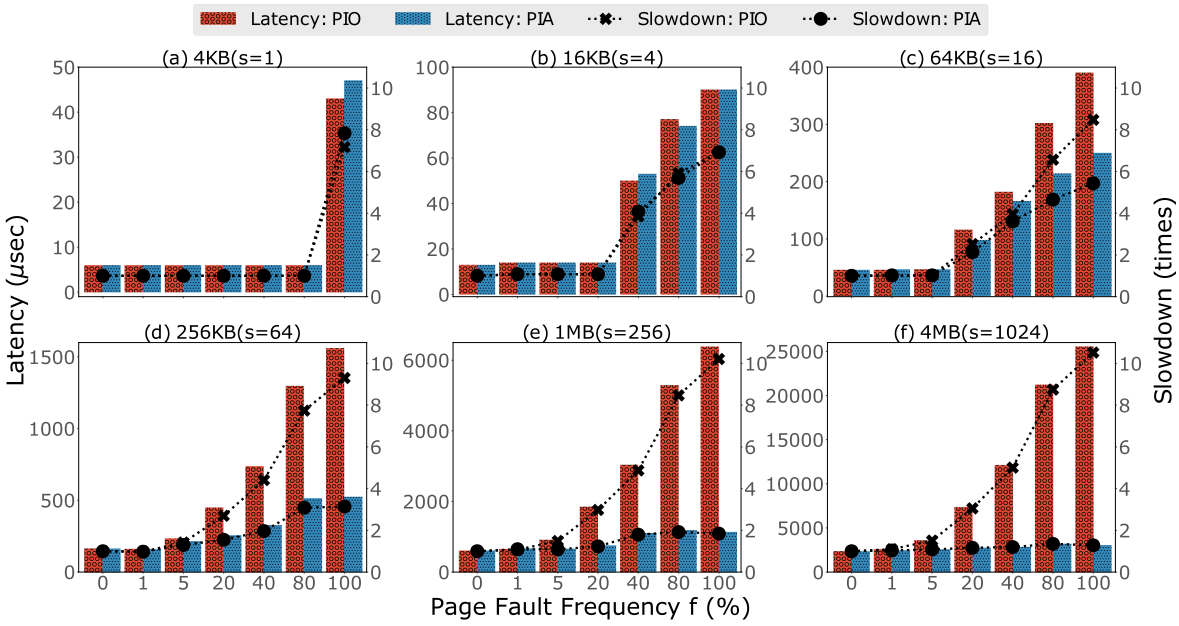


Fig. 14. Latency of handling page faults by PART and slowdown versus page fault frequency for various transfer sizes. Note: y-ranges differ.

(f=0%). We perform every experiment 5 times, for different random seeds, and report the average latency.

The main findings from Fig. 14 can be summarized as follows:

- Handling one page fault takes approximately 40 μsecs, whereas, the unit cost of (re)transmitting a 4KB page on a network link takes 3.2 μsecs (at 10 Gb/s). Effectively, the latency of PIO is approximately $\frac{40}{3.2}$ or nearly 12.5$\times$ higher than *Transfer-Only*. This is validated by our results especially for large transfers.
- The latency of PIO increases proportionally with page fault frequency f across all transfer sizes. This is expected, because in this scheme, the number of page faults and the number of retransmissions increase proportionally with the page fault frequency.
- With PIA, we resolve all imminent page faults immediately when the first page fails. Handling more pages using `get_user_pages()` takes time proportional to the number of pages of the transfer that fails, as seen for example in the 16KB case. According to the breakdown in Fig. 13, the overhead of `get_user_pages()` per failed page is approximately 3 μsec, but we drastically reduce the number of retransmissions and the times the handler is invoked. Effectively, for high page fault frequencies (e.g., 80 versus 100%), using PIA, we notice negligible difference in latency, especially for large transfers: the incremental cost of handling more page faults in a transfer is amortized by the transmission latency.
- PIA is considerably better than PIO especially for large transfers and high page fault frequencies. On the other hand, if we expect a few only random page faults, or the transfer size is small, PIO performs nearly on par with PIA.

Returning to the 4KB case, with frequency 100% (one page fault) PIA has slightly higher latency (2 μsecs) compared to PIO due to different code paths. For 16KB transfers, up to 4 pages can experience a page fault. PIA is expected to reduce the number of page fault interrupts. This benefit, however, is not evident for this transfer size, mainly because the transmission unit is 16KB (the full transfer in this case), and, also, because in this case PIO can resolve all faults with one tasklet invocation and one retransmission.

At the other extreme, for 4MB transfers, we notice that PIA performs considerably better compared to PIO, with a 7.1x improvement in latency. Also, compared to smaller transfer sizes, the slowdown latency for $f = 100\%$ is better, e.g., 1.5x for 4MB compared to 2.5x for 1MB and 6.2x for 64KB.

Take-Aways 1. When the transfer size increases, the overhead of handling page faults with PART , whether a few or all pages exhibit a page fault, is small, especially using PIA. *2.* The slowdown of PIA relative to the no-page-fault case (0%) decreases as we increase the transfer size.

6.1.2 Impact of Time Out Period

PART supports fast retransmissions upon a page fault (using *ERR* messages) and uses timeouts for resiliency. The default timeout value is 1ms, but in this section we examine different timeout values. As discussed in the previous section, for large transfers, the RDMA engine can timeout before the page-in tasklet, using PIA, proactively brings-in all pages in memory. Thus, instead of waiting for all pages to become present in memory and then launch the retransmission by sending the the ERR message, using a shorter timeout period, the source can start the retransmission after the first failed pages have been brought in memory. In the following experiment, we evaluate the performance of PART using the PIA approach for different RDMA timeout periods when all pages of the transfer exhibit a page fault.

The results are presented in Fig. 15. We examine three different timeout periods (TO): 1ms, 500 μs, and 100 μs. We also examine the performance if we disable the timeouts

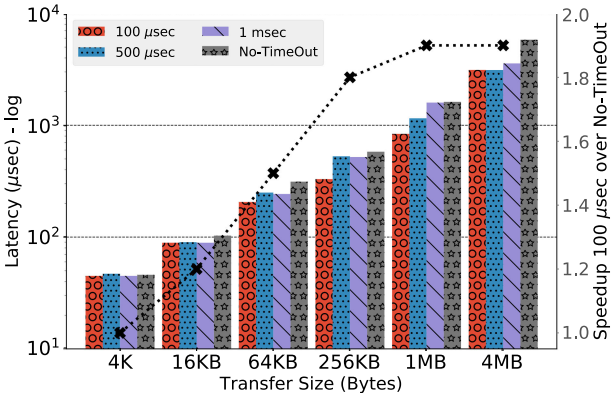


Fig. 15. Latency of page fault handling by PART for various transfer sizes under different timeout periods (100/500/1000μsec and NTO).

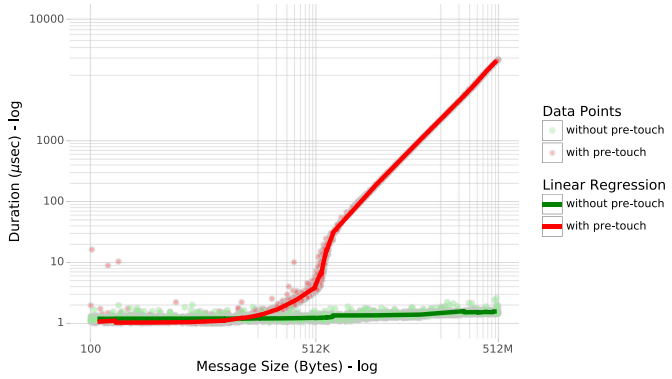


Fig. 16. MPI_Isend performance on a system that supports PART (no need to pre-touch buffers) versus a system that pre-touches buffers prior to each operation.

(NTO): when a first page experiences a page fault, a special NACK is sent to the source, which instructs the RDMA to cancel the timeout for this block, and wait for the ERR request, thus, no timeout is involved and consequently no overlap. We should note that Fig. 15 is very different from Fig. 11; Fig. 11 discusses performance of handling page faults using only timeout —no fast ERRs are involved. On the other hand, Fig. 15 covers the case where we handle page fault either in a combination of ERR + timeout, or only-ERR (and no timeout).

For transfer sizes up to 16KB, there is no significant difference in performance mainly because the transfers finish in time that is less than our smallest time-out period (100 μsecs). Starting from 256KB transfers, we notice that the 100 μsecs timeout period performs better than the other two timeout periods. The benefits of overlapping using 100 μsecs timeouts becomes even more visible for transfer sizes of 1MB and 4MB. However, in 4MB transfers we notice that the latency with TO equals to 100 μsecs and 500 μsecs is similar (yet they both outperform the case of TO=1ms).

On the other hand, if we disable the timeouts (No Time-Out, NTO), while increasing the transfer size by a factor of 4x, there is a 2-3x increase in latency. The NTO configuration also verifies our measurements in the (serial) breakdown of Fig. 13. In that case, we noticed that for 4MB transfers, the expected latency from the components of the breakdown was worse compared to the actual latency with 1ms timeout (5.7 ms versus 3.6 ms). Our current results validate that this was due to the overlap caused by the "early" timeout: the NTO configuration behaves as anticipated in the (serial) breakdown.

On the right y-axis of Fig. 15, we also depict the speedup of the 100 μsecs timeout versus NTO. As can be seen, the speedup increases with the transfer size, as expected, but saturates after 1MB transfer size, to a value of approximately 1.8×. Substituting $G(n)$ an $T(n)$ in Eq. (1), using our measurements from the previous section for the dynamic (unit) cost of page-in and transfer, as $G(n) = 3 \cdot n$ μsecs, $T(n) = 3.2 \cdot n$ μsecs (10G link), and taking the limit as $n \rightarrow \infty$, we get the speedup for large transfers, $s = 1.93$, independent of the TO value. This is very close to what we observe in the figure for the TO value of 100 μsecs.

We conclude that TO=100 μsecs provides the best latency in our setup. One might be tempted to test smaller timeout values, but 100μsecs is already small for a network featuring a few microseconds end-to-end latency. The problem is that a too low timeout value can cause spurious retransmissions when the network is loaded.

Take-Aways 1. Among all three different timeout periods examined, the smaller timeout of 100 μsec provides the best performance. The benefit of such a small timeout becomes evident especially for large transfers and large page fault frequency. This happens because the RDMA source engine starts sending data while the destination handlers continue to bring-in pages, thus introducing a beneficial overlap between page fault handling and transfer retransmission. *2.* Smaller timeout periods are beneficial for smaller transfer sizes: when the transfer size increases (e.g., 4MB), the overlap and thus the performance is more or less similar independently of the timeout value.

6.2 Real HPC Applications and Benchmarks

We also evaluated our mechanism by running MPI applications. We have ported MPI on top of our RDMA primitives [36]. MPI uses RDMA Read operations. So far, we described the path of an RDMA Write, because in our RDMA Engine the RDMA Read path is similar. To be more precise, whenever there is an RDMA Read, the RDMA Engine of the initiator node notifies the RDMA Engine of the target node, which now becomes responsible to execute a corresponding RDMA Write.

6.2.1 *Asynchronous MPI Calls*

Without pinning buffers, and without the capability of handling dynamic page faults, the user has to touch all pages prior to a transfer because the user typically does not know if a page is already in memory or not. One may expect that accessing buffers that are already in-memory may not induce a significant overhead overall. In the next experiment we use asynchronous MPI calls to evaluate the cost of pre-touch when buffers are already in memory.

In Fig. 16, we present the duration of asynchronous `MPI_Isend` calls both *with pre-touch* and *without pre-touch* of the buffers used. Following the `MPI_Isend` semantics, this duration includes all the overhead incurred by the preparation of the transmission from the library, until the flow returns to the user application, but not the transmission itself.

With pre-touch, we touch buffers once before the measurement starts. Then, during the measurement, we include the cost of just touching the first byte of each page–which is already in memory–, without including the overhead of the communication. In the case of *without pre-touch*, we never touch the buffers, and we again do not include the overhead of the communication. The measurements were obtained through a synthetic benchmark, which uses `MPI_ISend` for messages of different sizes. The message size for each transmission is chosen following a random uniform distribution in the range from 100 Bytes to 512 MB, with several iterations for each size.

In Fig. 16, we notice that *with pre-touch* all transfers pay a fixed overhead that increases as the transfer (buffer) size increases. This is explained because each touch requires a load and a store back to the first byte of each page of the buffer. On the other hand, *without pre-touch*, which would be the case of environments where PART is used, a fixed, insignificant cost is paid for all transfer sizes.

One may be tempted to optimize the pre-touch of buffers. For example, InfiniBand's ibv_advise_mr() allows pre-faulting of a given address range in a single system call. Alternatively, someone could use madvise with MADV_WILLNEED flag enabled. Another possibility is examined in [26], where the authors try to pre-touch buffers at the time of MPI rendezvous protocol.

Take-Away. In general, pre-faulting (also known as pre-touching) of buffers prior to an asynchronous send command in order to avoid page faults can lead to significant performance overheads, especially for large transfers. PART alleviates this problem by handling dynamically the occasional page faults, thus there is no need to pre-touch buffers.

6.2.2 LAMMPS

In the following experiments, we evaluate PART on real HPC applications. PART dynamically handles all page faults, versus a solution that touches all buffers prior to transfer, thus leading to no page faults (assuming that Transparent Huge Pages -THP- is disabled).

LAMMPS [41] is a Molecular Dynamics Simulator, which was ported to run in our testbed consisting of 16 nodes and 64 cores employing PART . For this experiment we use from 1 up to 16 parallel processes. The results depicted in Table 2 use four threads per process –this allows our system to handle multiple messages from different threads within a node, and different processes across nodes. In our experiments, we varied how many processes (thus FPGAs) participated in the run, ranging from 1 up to 16 (16 FPGAs/4 QFDBs/1 blade).

One of the main principles of PART is that not pinning pages and handling page faults at the network through retransmissions should not introduce any measurable overhead. In Table 2, we see that this is achieved when running LAMMPS, since the performance remains the same with page faults enabled (i.e., the case of *without pre-touch*) or not (i.e., the case of *with pre-touch*). More specifically, PART does not degrade performance by more than 1.1%. Also, we see that the standard deviation (std) of the Loop time is very small compared to the average reported.

TABLE 2
LAMMPS Performance (up to 1600 Steps) Using PART versus When we Page-In All Buffers Prior to the Transfer (PART Disabled)

	Processes	Loop time (sec)	Timesteps/s	Loop time (std)
No Page Faults (PART disabled)	1	76.47	1.31	0.29
	2	79.37	2.52	0.39
	4	84.74	4.72	0.75
	8	90.21	8.87	1.64
	16	98.88	16.20	3.45
With Page Faults (PART enabled)	1	76.44	1.31	0.46
	2	79.21	2.53	0.42
	4	84.55	4.73	0.70
	8	90.43	8.85	1.69
	16	99.95	16.03	4.14

We also measured the number of SMMU fault handler invocations. In these runs we found only a few SMMU fault handler invocations, 1489 in total for 16 processes, and all occurred at the beginning of the experiments. This happens because LAMMPS reuses its buffers that participate in communication. Note that due to interrupt coalescing in the OS we expect the number of page faults to be somehow higher than the number of the reported SMMU fault handler invocations, but we do not have means to measure them explicitly.

We also measured the number of ERR messages sent. This number depends on the number of processes as well, and for 16 processes we saw 239 ERRs in total.

In our HPC experiments with pre-touching, we have modified the MPI library so that pages are pre-touched exactly before they are used in RDMA transfers. Therefore, in these experiments, we expect marginal overhead on memory utilization when compared to PART –although we did not measure it explicitly–similar to the case where one pins the buffer prior to the transfer and unpins it after the transfer (pin-per-transfer strategy).[4] On the other hand, when a programmer resorts to an one-time pin strategy, such as pinning large buffers ahead of time in order to make sure that they are available for RDMA, the memory utilization can be affected, as shown in [25].

6.2.3 HPL and HPCG Benchmarks

High Performance LINPACK (HPL) [42] benchmark is used to solve a random dense linear system in double precision (64 bits) arithmetic on distributed memory systems. High Performance Conjugate Gradients (HPCG) Benchmark [43], [44] complements HPL in a way, by reaching a different and broad set of parallel applications. HPL is preferred for evaluating a subset of comparatively computationally-bound applications. HPCG is more memory- and node interconnect- performance dependent, thus it can better describe many non-computationally-bound workloads [45].

We evaluate PART running both HPL and HPCG benchmarks. In these experiments we use from 1 to 16 parallel processes. Tables 3 and 4 present the performance results for

4. Compared with pinning, pre-touching pays a system call only if the page is not in memory (i.e., a page fault).

TABLE 3
HPL Performance Using N = 16384 in 1 Rank With PART Being Enabled and Disabled

	Processes	Time (sec)	Performance (GFLOP/s)	Time (std)
No Page Faults (PART disabled)	1	939.57	3.12	2.51
	2	500.95	5.85	1.96
	4	280.97	10.44	0.66
	8	146.59	20.01	0.59
	16	79.06	37.09	0.35
With Page Faults (PART enabled)	1	940.34	3.12	2.01
	2	499.67	5.87	3.43
	4	281.78	10.41	1.09
	8	147.24	19.92	0.90
	16	81.47	36.00	1.37

HPL and HPCG, respectively, with PART being disabled (no page faults because of pre-faulting) and enabled (exhibiting and handling page faults). PART, which is triggered when the benchmark exhibits page faults during RDMA, does not degrade performance by more than 2.9%. Furthermore, we see that the standard deviation (std) of the latency in the tables is relatively small, in all scenarios, compared to the average reported.

We have collected additional statistics, regarding the number of MPI_Send() calls, the average number of Bytes per call, and the total Bytes sent using these calls per application. We report these numbers in Table 5 for all three HPC applications that were run using 16 nodes. As can be seen, the number of Bytes exchanged between nodes per run varies across the three applications. However, in all three applications, we believe there is a sufficient number of Bytes sent (and received) to stress our dynamic page fault handling mechanism. We measured 1497 and 318 SMMU fault handler invocations during RDMA faults for HPL and HPCG, respectively, when using for 16 processes. The number of ERRs depends on the setup (i.e., the number of processes); for 16 processes for HPL we noticed 220 ERRs per run, whereas for HPCG the corresponding number was 48.

From these results PART proves to be a good solution for HPC applications. It does not degrade the performance, it improves programmability (no need to pre-touch/pin buffers), and can also improve the memory utilization. When running real HPC applications, the overheads of dynamic page fault handling is very small ($< 1\%$ slowdown) mainly because (a) the per-page fault overhead (38 μsec for 4KB) is tolerable and further reduced for larger buffers thanks to our optimization, and (b) the re-usability of buffers in common HPC applications results in small page fault frequency.

TABLE 4
HPCG Performance Using 128 128 128 (Each MPI Process was Computing a Solution for a Cube of Size of 128 Points) 60s Timed Portion in 1 Rank With PART Being Enabled and Disabled

	Processes	Time (sec)	Performance (GFLOP/s)	Time (std)
No Page Faults (PART disabled)	1	356.71	0.11	6.12
	2	350.49	0.22	4.06
	4	357.10	0.43	5.45
	8	356.59	0.87	4.36
	16	361.74	1.72	3.84
With Page Faults (PART enabled)	1	345.38	0.11	9.36
	2	340.50	0.23	5.86
	4	348.82	0.45	5.40
	8	350.60	0.89	2.37
	16	356.25	1.75	4.36

TABLE 5
MPI_Send() Traffic of Applications

Application	MPI_Send () calls	Total MBytes Transferred	Average KBytes per MPI_Send()
LAMMPS	2107644	31035	15.1
HPL	176404	7388	42.9
HPCG	267408	6041	23.1

7 Conclusion

In this paper, we proposed, optimized and evaluated PART, a novel end-to-end mechanism that resolves occasional page faults during RDMA. PART leverages the retransmission capabilities of modern NICs. PART re-uses the IOMMU in order to perform virtual-to-physical translations during user-level initiated RDMA transfers; thus, no separate memory management framework is required, in contrast to modern NICs.

PART can handle RDMA page faults while servicing other memory requests. Thanks to this overlap, resolving dynamic page faults with PART performs better than pinning pages in advance, especially for large transfers (1.46x better for 4MB). In summary, PART unlocks the advantages of dynamic paging, which includes enhanced memory utilization, simplifies the programming model, and can perform better than pinning.

Acknowledgments

We would like to thank all the reviewers for their valuable feedback. Furthermore, we are grateful for the insightful comments and feedback provided by Prof. Panagiota Fatourou on this article. We would also like to thank our collaborator Astrinos Damianakis for his support in running experiments for the revised version of this work.

References

[1] E. P. Markatos and M. G. H. Katevenis, "User-level DMA without operating system kernel modification," in *Proc. 3rd Int. Symp. High-Perform. Comput. Archit.*, 1997, pp. 322–331.

[2] E. V. Carrera, S. Rao, L. Iftode, and R. Bianchini, "User-level communication in cluster-based servers," in *Proc. 8th Int. Symp. High-Perform. Comput. Archit.*, 2002, pp. 275–286.

[3] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *Proc. 11th USENIX Conf. Netw. Syst. Des. Implementation*, 2014, pp. 401–414.

[4] Y. Zhu *et al.*, "Congestion control for large-scale RDMA deployments," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2015, pp. 523–536.

[5] A. Tavakkol *et al.*, "Enabling efficient RDMA-based synchronous mirroring of persistent memory transactions," 2018, *arXiv: 1810.09360*.

[6] S. Hu *et al.*, "Deadlocks in datacenter networks: Why do they form, and how to avoid them," in *Proc. 15th ACM Workshop Hot Topics Netw.*, 2016, pp. 92–98.

[7] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 295–306.

[8] C. Jia *et al.*, "Improving the performance of distributed TensorFlow with RDMA," *Int. J. Parallel Program.*, vol. 46, no. 4, pp. 674–685, 2018.

[9] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," *Int. J. Parallel Program.*, vol. 32, no. 3, pp. 167–198, 2004.
[10] C. Bell and D. Bonachea, "A new DMA registration strategy for pinning-based high performance networks," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2003, pp. 10.
[11] Y. Ajima *et al.*, "The Tofu interconnect D," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2018, pp. 646–654.
[12] A. Kalia, M. Kaminsky, and D. Andersen, "Design guidelines for high performance RDMA systems," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 437–450.
[13] K. Taranov, R. Bruno, G. Alonso, and T. Hoefler, "Naos: Serialization-free RDMA networking in Java," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/taranov
[14] M. Vardoulakis, G. Saloustros, P. G. Férez, and A. Bilas, "Using RDMA for efficient index replication in LSM key-value stores," 2021, *arXiv:2110.09918*.
[15] N. Binkert, A. Saidi, and S. Reinhardt, "Integrated network interfaces for high-bandwidth TCP/IP," in *Proc. 12th Int. Conf. Architect. Support Program. Lang. Oper. Syst.*, 2006, pp. 315–324.
[16] G. Liao, X. Znu, and L. Bnuyan, "A new server I/O architecture for high speed networks," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 255–265.
[17] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler, and W. Rehm, "Analysis of the memory registration process in the Mellanox InfiniBand software stack," in *Proc. 12th Int. Conf. Parallel Process.*, 2006, pp. 124–133.
[18] F. Mietke, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm, "Reducing the impact of memory registration in InfiniBand," in *Proc. KiCC Workshop*, 2005.
[19] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa, "Pin-down cache: A virtual memory management technique for zero-copy communication," in *Proc. 1st Merged Int. Parallel Process. Symp. Symp. Parallel Distrib. Process.*, 1998, pp. 308–314.
[20] P. W. Frey and G. Alonso, "Minimizing the hidden cost of RDMA," in *Proc. IEEE 29th Int. Conf. Distrib. Comput. Syst.*, 2009, pp. 553–560.
[21] L. Liss, Y. Birk, and A. Schuster, "In-kernel integration of operating system and infiniband functions for high performance computing clusters: A DSM example," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 9, pp. 830–840, Sep. 2005.
[22] C. Guo *et al.*, "RDMA over commodity ethernet at scale," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 202–215.
[23] R. Neugebauer, G. Antichi, J. Zazo, Y. Audzevich, S. López-Buedo, and A. Moore, "Understanding PCIe performance for end host networking," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 327–341.
[24] S. Novakovic *et al.*, "Storm: A fast transactional dataplane for remote data structures," in *Proc. 12th ACM Int. Conf. Syst. Storage*, 2019, pp. 97–108.
[25] I. Lesokhin *et al.*, "Page fault support for network controllers," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2017, pp. 449–466.
[26] M. Li, K. Hamidouche, X. Lu, H. Subramoni, J. Zhang, and D. K. Panda, "Designing MPI library with on-demand paging (ODP) of InfiniBand: Challenges and benefits," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 433–443.
[27] M. Li, X. Lu, H. Subramoni, and D. K. Panda, "Designing registration caching free high-performance MPI library with implicit on-demand paging (ODP) of InfiniBand," in *Proc. IEEE 24th Int. Conf. High Perform. Comput.*, 2017, pp. 62–71.
[28] M. Technologies, "Understanding on demand paging (ODP)," 2019. [Online]. Available: https://community.mellanox.com/s/article/understanding-on-demand-paging–odp-x
[29] T. Fukuoka, S. Sato, and K. Taura, "Pitfalls of InfiniBand with on-demand paging," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2021, pp. 265–275.
[30] Kernel.org, "Shared virtual addressing (SVA) with ENQCMD," Version v5.17, Mar. 2022. [Online]. Available: https://www.kernel.org/doc/html/v5.17/x86/sva.html
[31] Y. Xie and B. Liu, "Share virtual address," 2018. Accessed: Mar. 20, 2022. [Online]. Available: https://static.sched.com/hosted_files/lc32018/e0/LinuxCon18_Share_Virtual_Address.pdf
[32] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel - From I/O Ports to Process Management: Covers Version 2.6)* 3rd ed. Sebastopol, CA, USA: O'Reilly, 2005.
[33] W. Cohen, "Examining huge pages or transparent huge pages performance," 2014. Accessed: Mar. 20, 2022. [Online]. Available: https://developers.redhat.com/blog/2014/03/10/examining-huge-pages-or-transparent-huge-pages-performance/
[34] J. Corbet, "Transparent huge page reference counting," Nov. 2014. Accessed: Mar. 20, 2022. [Online]. Available: https://lwn.net/Articles/619333/
[35] R. Ammendola *et al.*, "The next generation of Exascale-class systems: The ExaNeSt project," in *Proc. Euromicro Conf. Digit. Syst. Des.*, 2017, pp. 510–515.
[36] M. Ploumidis *et al.*, "Software and hardware co-design for low-power HPC platforms," *Proc. Int. Conf. High Perform. Comput.*, vol. 11887, pp. 88–100, 2019.
[37] Xilinx, "Zynq UltraScale+ MPSoC tech. reference manual UG1085," Version 2.2, 2020. Accessed: Mar. 20, 2022. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1085-zynq-ultrascale-trm
[38] ARM, "ARM system memory management unit architecture specification - SMMU architecture version 2.0," 2016, [Online]. Available: https://developer.arm.com/documentation/ihi0062/latest
[39] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov, "Linux netlink as an IP services protocol," *RFC*, vol. 3549, pp. 1–33, 2003, doi: 10.17487/RFC3549.
[40] R. Mittal *et al.*, "Revisiting network support for RDMA," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 313–326.
[41] P. Steve, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, no. 1, pp. 1–19, 1995.
[42] A. Petitet, R. Whaley, J. Dongarra, and A. Cleary, "HPL—A Portable implementation of the high-performance Linpack benchmark for distributed-memory computers," Version 2.3, 2018. Accessed: Mar. 20, 2022. [Online]. Available: https://www.netlib.org/benchmark/hpl/
[43] J. Dongarra and M. Heroux, "Toward a new metric for ranking high performance computing systems," Sandia Nat. Lab., Albuquerque, NM, USA, Tech. Rep. SAND2013–4744, Jun. 2013.
[44] M. Heroux, J. Dongarra, and P. Luszczek, "HPCG technical specification," Sandia Nat. Lab., Albuquerque, NM, USA, Tech. Rep. SAND2013–8752, Oct. 2013.
[45] P. Gschwandtner, A. Hirsch, P. Thoman, P. Zangerl, H. Jordan, and T. Fahringer, "The cluster coffer: Teaching HPC on the road," *J. Parallel Distrib. Comput.*, vol. 155, pp. 50–62, 2021.

Antonis Psistakis received the BSc degree in computer science and the MSc degree in computer science and engineering from the University of Crete (UoC), Heraklion, Crete, Greece, in 2017 and 2019, respectively. He is currently working toward the PhD degree with the University of Illinois at Urbana-Champaign (UIUC), Urbana, Illinois. His research interests include parallel and distributed systems and computer architecture. He is a member of UIUC, and the i-acoma Group, Urbana, IL, USA.

Nikos Chrysos currently working toward the PhD degree with Foundation for Research & Technology – Hellas (FORTH), working on interconnection networks for high-density, Exascale-class datacenters, and high-performance computers, as well as on virtualized heterogeneous platforms. From 2009 to 2014, he was with IBM Research where he contributed to the design and implementation of high-performance server-rack fabrics for 100G Ethernet. He is the author or co-author of more than 16 granted patents.

Fabien Chaix received the engineering degree from INP-Grenoble, in 2008, the master's degree from the Université Joseph Fourier, in 2008, and the PhD degree from Grenoble Universities, in 2013. He is now a research fellow with FORTH, where he follows his research interests in high performance interconnects and resiliency.

Marios Asiminakis received the MEng degree in electronics and computer engineering from the ECE Department, Technical University of Crete (TUC), Chania, Crete, Greece, in 2017. He is a research engineer with the Computer Architecture and VLSI Systems (CARV) Laboratory - Institute of Computer Science (ICS) - Foundation for Research and Technology Hellas (FORTH), Heraklion, Greece. He has worked as a key system software developer in the European projects ExaNeSt and EuroExa. His research interests include system software for high performance computing.

Michalis Gianioudis received the BSc degree in computer science from the Computer Science Department, University of Crete, in 2017. Since 2017, he has been working as research hardware engineer with the Computer Architecture and VLSI Systems (CARV) Laboratory - Institute of Computer Science (ICS) - Foundation for Research and Technology Hellas (FORTH), Heraklion, Greece. He has worked in the development of European projects such as EUROSERVER, ExaNeSt, ExaNode, and EuroExa. His research interests include HPC hardware architectures.

Pantelis Xirouchakis received the BSc degree in physics from the Physics Department, University of Crete (UoC), Heraklion, Crete, Greece, in 2016, and the MSc degree in computer science and engineering from the Computer Science Department, University of Crete, Heraklion, Crete, Greece, in 2019. He is a research engineer with the Computer Architecture and VLSI Systems (CARV) Lab, FORTH-ICS, Heraklion, Crete, Greece. He has worked as a key FPGA developer in the European projects ExaNeSt, ExaNode, and EuroExa. His research interests include hardware architectures for high performance computing.

Vassilis Papaefstathiou received the PhD degree in computer science from the University of Crete, in 2013. He is a researcher with FORTH-ICS. From 2001 to 2003, he worked on IC design and verification in ISD S.A. and collaborated closely with STMicroelectronics on industrial SoC designs. From 2005 to 2013, he was a research engineer with the CARV Lab, ICS-FORTH. From 2014 to 2016, he was a postdoctoral researcher with the Chalmers University of Technology, Sweden. Since September 2016, he is with FORTH. He has been heavily involved in several EU-funded research projects (EPI, eProcessor, EuroEXA, ExaNest, ExaNoDe, ECOSCALE, EuroServer, ERC MECCA, SHARCS, ENCORE, SARC, UNISIX, SIVSS) and has designed several FPGA-based hardware prototypes for multicore architectures and high-performance interconnects. He is currently coordinating FORTH's tasks in EPI and eProcessor and leads the hardware development team. His research interests include parallel computer architecture, high performance computing, high-speed interconnects, low-power datacenter servers, and storage systems, with particular emphasis on cross-layer design and optimization.

Manolis Katevenis received the PhD degree from U.C. Berkeley, in 1983, and the ACM Doctoral Dissertation Award, in 1984 for his thesis on "Reduced Instruction Set Computer (RISC) Architectures for VLSI". After a brief term on the faculty with Stanford University, he has been with the University of Crete and with FORTH, as a professor of computer science now. After RISC, his research has been on interconnection networks and interprocessor communication.