PART: Pinning Avoidance in RDMA Technologies

Antonis Psistakis, Nikos Chrysos, Fabien Chaix, Marios Asiminakis,
Michalis Giannioudis, Pantelis Xirouchakis, Vassilis Papaefstathiou, Manolis Katevenis
Institute of Computer Science
Foundation for Research and Technology – Hellas (FORTH)
Herakleion, Crete, Greece
{psistakis, nchrysos, chaix, marios4, yanoudis, pxirouch, papaef, kateveni}@ics.forth.gr

***Abstract*—State-of-the-art Remote Direct Memory Access (RDMA) engines pin communication buffers, complicating the programming model, limiting the memory utilization, and mandating a separate memory translation subsystem spanning the network interface card and the OS. In this paper, we introduce PART, a page fault handling mechanism suitable for emerging nodes that integrate the NI with the main processor. PART does not need to pin pages, thus any process buffer can be used for communication, and resolves occasional page-faults dynamically, when the network accesses the memory, by reusing the RDMA transport. Additionally, PART leverages the I/O Memory Management Unit (IOMMU) which is next to the processor in order to translate virtual to physical addresses, thus reducing cost and complexity. We implement and evaluate PART in a cluster of 16 nodes and 64 ARM cores. We evaluate the performance of transfers for varying page fault frequency, and examine optimizations that proactively page-in all pages upon the first page fault or ahead of the transfer, providing useful insights that can be used to optimize runtimes. Our results show that PART completes one-page transfers with a minor page-fault at the destination in approximately 38 μsecs, while the slowdown on 1MB transfers that experience faults in all pages is as little as 2.6x compared to the no-page-fault case. Page faults are expected to be rare in HPC setups: the performance of LAMMPS in our cluster is virtually unaffected when pages are handled dynamically using PART.**

***Index Terms*—page faults, RDMA, IOMMU, MPI, low-power ARM processors**

I. Introduction

Modern computing systems strive to eliminate the use of the kernel path in processor communication [1]–[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. 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 by the RDMA subsystem in order to fetch and write data to memory. Common RDMA technologies copy page mappings into Network Interface Cards (NICs), which become responsible for address translation. These NICs cannot tolerate page faults, thus, in order to circumvent them, the Operating System (OS) pins the memory pages that correspond to communication buffers [6]. Pinning is undesirable for the following reasons:

1) Extensive use of pinning can hinder the memory utilization [7] and is not compatible with some optimizations of the OS (e.g. Transparent Huge Pages).
2) Pinning and unpinning pages requires system calls that introduce overheads.
3) The applications are responsible to pin and unpin their working set of communication buffers, rendering programming more difficult.

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

Following these insights, we present PART, a system that does not pin the communication buffers. In PART, we treat occasional page-faults similarly with other transmission errors that may occur in a transfer. In a nutshell, we first resolve the page-fault, locally, at the node that it occurs, and subsequently we re-transmit the failing pages of the RDMA transfer. We implement PART in a cluster of low-power ARM processors. For address translation, PART utilizes the existing IOMMUs, thus obviating the cost and overheads of translation-capable NICs. Our contributions in this paper are the following:

- We propose and evaluate PART, a system that handles the page-faults during RDMA by retransmitting failed pages, thus removing the need to pin buffers in memory.
- We assume that the NI is closely coupled with the processor, and we re-use the IOMMU for address translation instead of relying on specialized NICs.
- We implement both the system software and hardware components of PART in small cluster of interconnected ARM processors.
- We run LAMMPS on 16 nodes and 64 cores. Our results show that PART performs virtually as well as a system that avoids network page-faults. In microbenchmarks, we vary the frequency of page faults and discuss various optimizations and trade-offs.

This work was supported by the European Commission under the Horizon 2020 Framework Programme (Grant Agreement 671553 and 754337)

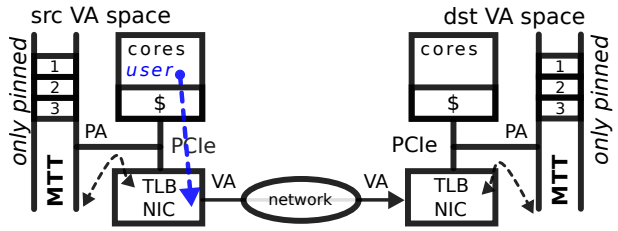

Fig. 1: Pinning in traditional network interface cards (NICs). Pages are pinned prior to the transfer, and the mappings are copied in a separate memory subsystem that spans the NIC and main memory.

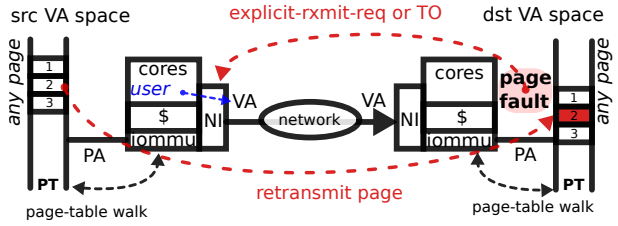

Fig. 2: PART's general architecture. In PART, we do not pin pages, and memory translations go through the IOMMU, and the page tables (PT) of the user processes. Page-faults are handled by retransmitting failed pages after a time-out, or explicit retransmit messages.

The remainder of this paper is organized as follows. In Section II, we present the motivation for this work, related work, and outline PART. Next, in Section III, we describe the processor and the network interface that we use to implement PART. Section IV describes our page-fault handling mechanism, and Section V presents our results. Finally, we conclude with Section VI.

II. Motivation and Overall Approach

Common RDMA technologies register the address mappings of pages that participate to communication in special Memory Translation Tables (MTTs) in host memory. To keep the mappings of registered pages coherent with changes happening in the page table maintained by the OS, these pages are pinned so that their physical locations (frames) do not change while the network accesses them.

This approach has significant implications. First, registering or de-registering a page is a privileged task, involving costly system calls. An early study [8] measured the pin-down overhead for Myrinet networks as $(40 \times 7 \cdot n)$ μsecs, where n is the number of pages. Later, in [9], the total cost of registering a page was found approximately 30 μsecs; this cost stays constant for buffers up to 128KB, and increases proportionally with the buffer size afterwards. In our newer ARM-based platform, we measured the latency of pinning a page, and we found it is approximately 6 μsecs.

Applications that use RDMA for improved performance wish to hide the registration latency. Special memory management techniques for communication buffers have been proposed for that purpose. For instance, lazy de-registration in [8] defers releasing pages anticipating future re-use. In [6], the authors propose to copy small messages in pre-pinned buffers, whereas other studies propose to register the entire physical address space [10]. Finally, large page sizes and modern NIC optimizations for contiguous physical memory may help even further [11].

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. The authors in [12] report that the cache of the MTT in the NIC of their study has at most 2K entries. For 4KB pages, this cache can keep translations for 8MB 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 (300-500 nsecs one-way latency in unloaded systems). In high-speed networks, this dragging path can induce a "slow receiver syndrome", which can cause and spread congestion throughout the network.

Most of the aforementioned solutions hinder the resource management, scalability and latency benefits of "Demand Paging", because communication pages have to be resident in memory prior to actual access. Furthermore, these solutions typically maintain a secondary memory translation subsystem for communication buffers, which increases cost and complexity.

A previous work, independent of ours, resolves network page faults dynamically [13]. The overhead of that mechanism for 4KB messages is approximately 220 μsecs, whereas in PART the corresponding overhead is approximately 38 μsecs. In [13], the authors comment that 90% of this cost is due to slow NIC firmware. Additionally, in their work they use special translation tables in the NIC and the host memory, whereas PART uses the existing IOMMU and the process page table. Finally, PART resolves all page faults by re-using the existing RDMA transport and utilizes optimizations such as proactively paging-in all pages of the RDMA transfer.

In this paper, we propose and evaluate PART, a mechanism suitable for next-generation systems, featuring *a network interface tightly coupled with the main processor and the host memory*. This arrangement bypasses the overheads of PCIe, thus enabling faster access between the network interface (NI) and the host memory as well as between the NI and the processor. In these emerging architectures that integrate the NI in the same chip or package with the main processor, it is important to re-use existing blocks.

PART does not pin pages before using them for RDMA. Effectively, PART can *use any virtual address for communication*, but 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 Figure 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 NI and the OS, as is the case for current NICs, shown in Figure1.

In our ARM-based testbed, the latency of a small transfer incurring a network page fault is approximately 38 μsecs, as detailed in Section V. This includes (and is dominated by) the time needed to wake up from an asynchronous page-fault and to resolve the page-fault. Even though the page-fault latency may exceed the overhead of page pinning, *page faults are rare for well-designed HPC applications* that fit their working set in memory in order to avoid random latencies and achieve good synchronization, as well as for *in-memory data analytics* that maintain and process data in memory.

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.

III. Network interface re-using ARM's SMMU

In this section, we describe the ARM-based platform that we used to evaluate PART. The basic block is a Zynq MPSoC from Xilinx [14], four (4) ARMv8 low-power cores (A53), 16GByte DDR4, a rich set of hardware IPs, and reconfigurable logic, as shown in Figure 3. The ARM cores inside the MPSoC are fully coherent with each other, whereas the programmable logic can access memory through IO-coherent ports.

A. Lean RDMA transport

A custom *network interface (NI)* inside the FPGA part of the MPSoC implements a packet-based network protocol for inter-chip communication. The NI is highly virtualized, offering multiple channels that can be allocated to concurrent user-level processes. Processes initiate RDMA (write and read) transfers using virtual addresses, bypassing completely the OS. 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. The NI offers a reliable transport, based on end-to-end positive and negative acknowledgments and retransmissions, allowing transfers to complete without any software or kernel overhead.

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 blocks to the hardware. 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 starting from the destination VA. For each correctly completed block, a positive ACK is generated, which is routed to the co-processor

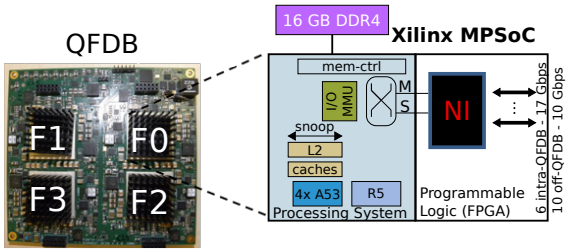

Fig. 3: PART's building blocks: (left) Quad-FPGA Daughter Board (QFDB) with 4 Zynq MPSoCs, described in Section V; (right) Each MPSoC provides 4 ARMv8 cores, 16GB DDR4 (connected with a parallel bus at 160 Gb/s), an IOMMU, and a custom RDMA transport (NI) implemented in the FPGA.

at the source. Missing ACKs trigger time-outs and block-level retransmissions.

Accesses to host memory coming from the NI pass through the ARM's IOMMU (SMMU), which translates virtual to physical addresses, as described 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. The hardware end-to-end latency is below 1 μsec; the R5 co-processor adds approximately 3 μsecs.

B. ARM's System Memory Management Unit

Instead of using a special NIC, PART makes use of the System Memory Management Unit (SMMU) [14] that is mainly responsible to manage the memory requests from I/O devices to the local memory of the system. The SMMU is defined by ARM, but, operation-wise, is similar to other IOMMUs. Virtual addresses are used in RDMA for virtualization and protection purposes. 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 that 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.

In addition, the SMMU includes Translation Lookaside Buffers (TLBs) that keep the most recent address translations, without needing to perform a page table walk. 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.

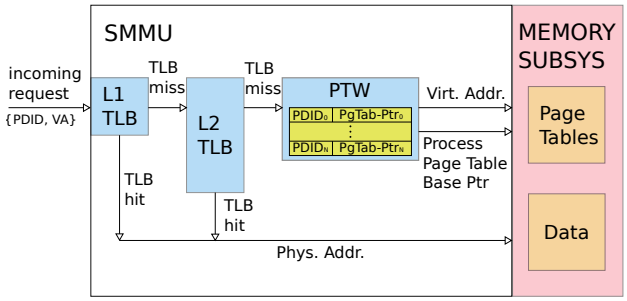

Fig. 4: Memory translations through ARM's SMMU subsystem

In Figure 4 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 *Page-Table Walk (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.

IV. PART: Handling RDMA page faults

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 be "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. In this paper, we present how PART treats such minor page-faults in the destination buffer. Nevertheless, PART can also handle page faults at the source, which are possible when internal optimizations (such as Transparent Huge Pages) in Linux are enabled.

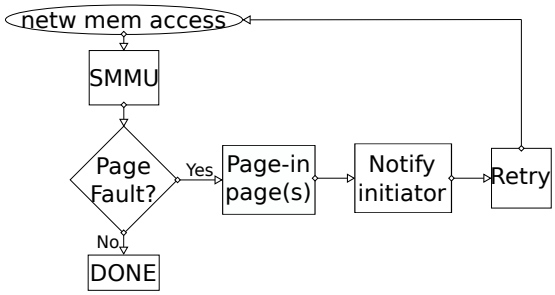

Fig. 5: General flow of PART

Figure 5 depicts the general flow of our mechanism. Each remote memory access to a computing node reaches first the (remote) RDMA Engine, then passes through the remote SMMU, and finally, if allowed, accesses the remote memory. Each page fault from a memory transaction that passes through SMMU invokes a particular fault handler. This fault handler catches all the faults generated by accesses from the SMMU and is defined in the ARM SMMU Linux driver (version 4.9) [15]. Each context bank, which corresponds to the page table of a process, has its own set of fault registers, that provide information about the cause of the fault, including the virtual address that triggered it. The original version of this handler only catches such asynchronous page faults[1] that were triggered by an unsuccessful SMMU page table walk, but does not provide support to handle them. We have enriched the functionality of this handler in order to resolve page faults.

Precisely, 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, so that the transfer will succeed after the retransmission. One way that this can be accomplished is to have a user-space thread, linked with the user-process executable, be informed about the fault, so it touches (i.e. reading and then writing the first byte), the failed page. The same can be accomplished in kernel-space, utilizing the `get_user_pages()` and `put_page()` methods of the kernel. The first essentially pins-in a range of pages, thus bringing them also in memory, and the latter unpins them. In PART, we opted for the kernel-based solution, because it avoids the utmost context switches from kernel space to user space, and, also, because it enables to *proactively page-in all pages*, an optimization that we discuss next.

Once we encounter a page fault on an RDMA transfer, PART can *proactively page-in more than one page* of the transfer. In Section V, we evaluate the performance of *Page-in 1-Pg*, which pages in only the failed page, *Page-in 4-Pgs*, which proactively pages-in all (4) pages in a block, and, finally, *Page-in All-Pgs* which pages-in all pages of an RDMA transfer, starting from the failed one. This requires that the mechanism knows the RDMA transfer size. On the other hand, get_user_pages() allows us to touch-ahead pages even when this information is not available.

When a page fault occurs, either at the source or at destination, the source DMA Engine will not receive an acknowledgment, and will retransmit (in hardware) the failed block after a time-out. The time-out of an RDMA Engine cannot be set at a meager value, because this can induce early time-outs and duplicates. Thus, in a network with a few microseconds end-to-end latency, the time-out may be set at 100 μsecs or more. Retransmission caused by failing packet CRC can be expedited using negative acknowledgments (NACKs). However, we cannot send such a NACK immediately when a page fault occurs, because the retransmission may arrive before we have paged-in the missing page(s). Instead, we send an *Explicit Retransmission Request (ERR)* (the equivalent of NACK), after the page-in task has completed its work. In our implementation, 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 Section V), in order to avoid multiple retransmissions of the entire transfer, which can occur when multiple pages fail.

RDMA page fault at the destination: Figure 6 depicts how PART handles a page fault at a destination address. The RDMA Engine receives a network packet at the destination

[1]These page-faults are not triggered by processor load or store commands, but by an asynchronous device memory access, or, in our case, by an RDMA transfer.

and tries to write data to local memory using ARM's AXI memory interconnect. The corresponding AXI write request has an (IO) virtual address which is translated by the SMMU before accessing the local memory. As can be seen in the Figure, the request generates a page fault, which triggers an interrupt that is catched by the SMMU fault handler. In parallel, an AXI negative acknowledgment (NACK) arrives at the local RDMA Engine, which logs in a NI FIFO all the necessary information needed in order to resolve the page fault and to request from the sender later to retransmit the failed block. When the AXI NACK arrives, a page-fault NACK is sent immediately to the source RDMA Engine, that will cancel the time-out retransmission.

As shown in Figure 6, 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, using `get_user_pages()` and `put_page()`, as discussed previously, and then triggers an ERR to the sender.

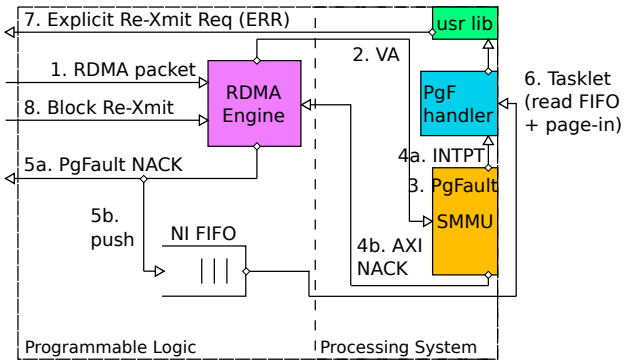

Fig. 6: Handling an RDMA page fault at the destination address.

RDMA page fault at the source: An RDMA page fault at the source address (buffer) is triggered during a local (AXI) read access to memory. Page faults at the source buffer during an RDMA transfer are not expected to be as common as page faults at the destination buffer. Because of that, in our current implementation, page faults at the source addresses are recovered after the expiration of a time-out period. This gives our page fault handling mechanism a sufficient time window to solve the page fault, using the same mechanisms as with page-faults at the destination.

V. Performance Evaluation

In this section, we evaluate PART using microbenchmarks and a real HPC application. In Table I, we see all the acronyms/types of measurement and their definition, that will be used for the rest of the paper. Our testbed consists of blades, also named Mezzanines, each embedding four (4) Quad-FPGA Daughter Boards (QFDBs), first shown in Figure 3. Each QFDB consists of 4 Xilinx Zynq MPSoCs, 64GB of DDR4 SDRAM (16GB per MPSoC), and High Speed Serial (HSS) links for external connections. The MPSoCs inside the QFDB are connected in an all-to-all fashion using 17 Gb/s HSS links. Inside the blade, the QFDBs are connected in an all-to-all fashion using 10 Gb/s HSS links. The latency of each hop in our prototype is approximately 150 nsecs, and the base (min) latency of RDMA operations is around 4 μsecs.

Every experiment is repeated between 5 and 50 times; unless otherwise noted, we report the average of the individual measurements. Measurements were conducted using software profiling; for userspace we used clock_gettime() (with CLOCK_MONOTONIC configuration) and for kernelspace we used ktime_get().

TABLE I: Description of methods

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 network page fault, page-in only one page
PI4 (Page-In 4-Pages)	Upon network page fault, page-in up to block-size number of pages
PIA (Page-In All-Pages)	Upon network page fault, page-in all remaining pages of the transfer

Ideal execution of an RDMA transfer: In Figure 7, 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

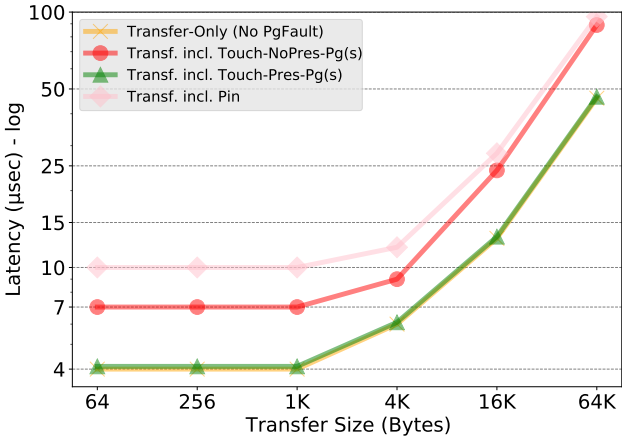

Fig. 7: Latency of RDMA (no page fault).

pages are pinned before the transfer, using the `mlock` system call. As can be seen, pinning adds 6 μsecs for small transfers (10 vs 4 μsecs) and the overhead increases with the transfer size, i.e. with the number of pages pinned, nearly doubling the latency of the operation.

Figure 7 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. Pages can either *be 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*, 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* (3 vs 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.

The added latency of touching pages before RDMA transfers becomes increasingly important in *asynchronous* transfers, such as *MPI_Isend*. 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.

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). But, as we discuss later, we can combine touch with PART, especially for small, synchronous transfers.

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 Figure 8, 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 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 in the NI FIFO (up to 16 read commands in our setup), pages-in the faulty pages, and communicates this information to userspace using Netlink sockets. After that, an ERR is issued from user-space to the initiator, which adds 1 μsec latency. Finally, retransmitting the missing page adds 6 μsecs, for a total latency of 31 μsecs.

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

Fig. 8: Breakdown for a 4KB RDMA Write transfer with page fault at destination.

In Figure 9, we measured 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 (Explicit Retransmission Requests), and, second, waiting for the source RDMA Engine to time-out (100 μsecs and 1 msec). As can be seen in Figure 9, when using ERRs the total latency of the 4KB RDMA Write transfer is approximately 38 μsecs, i.e. 7 μsecs higher than what we measured in our breakdown. We believe that this discrepancy is mostly due to context switches, the overhead of which is not included in Figure 8. Furthermore, Figure 9 shows that the latency is similar for transfers up to 4KB, and that ERR is much more efficient than waiting for the time-out. Nevertheless, RDMA Engines that do not support ERR can resort to time-outs in order to handle occasional page faults.

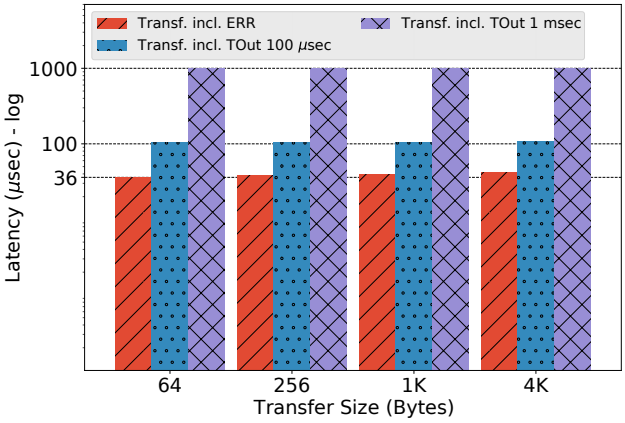

Fig. 9: Latency of RDMA with page fault at destination for different retransmission methods.

Proactively paging ahead upon page-fault: In Figure 10, we examine the latency of large RDMA transfers, in the extreme case *with page fault on all pages*. For comparison, we also include configurations from Figure 8, with no network page-faults, but now the tests extend for up to 1MB transfers.

Page-in 1-Pg (PIO) presents the latency of the transfer when PART handles every page-fault independently. As mentioned in Section III, our RDMA Engine splits every transfer into blocks of 16KB (4 pages of 4KB), and selectively retransmits

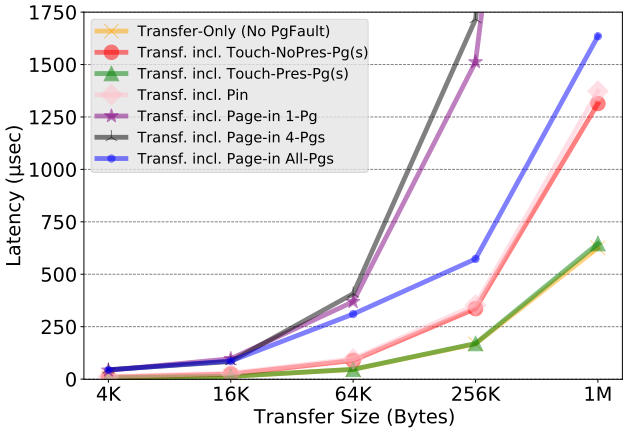

Fig. 10: Latency of an RDMA transfer with page faults in all destination pages.

entire blocks. In Figure 10 we also evaluate "Page-in 4-Pgs" (PI4), which, in every page-fault, pages in 4 pages that belong to the same block, proactively resolving all possible page-faults in that block with one retransmission. As can be seen, PI4 does not improve performance significantly. This can be explained because PIO triggers retransmissions faster, and because of the overlap among the operations that handle different pages in the same block; thus, even with PIO, all pages in a block can be present in memory when the retransmitted block arrives.

Going one step further, we also examine "Page-in All-Pgs" (PIA), an optimization of PART that proactively pages in all (forthcoming) pages of a transfer upon the first page fault. This scheme reduces the overheads and the latency especially for large transfers. Compared to PIO, proactively touching all pages presents the same latency for transfer sizes up to 64KB (16 pages). However, for larger transfers, the benefits become huge. PIA minimizes the number of retransmissions (1 block for the entire transfer), since all subsequent blocks of the transfer will succeed (pages will reside in memory).

The performance of PIA is worse than "Touch NoPres-Pgs". More precisely, as can be seen in Figure 10, 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. Both methods incur the latency of paging-in all pages (i.e. normal CPU MMU minor page faults), but PART does that from the tasklet, which can be interrupted multiple times due to SMMU handler invocations. On the other hand, as mentioned earlier, PART touches the pages dynamically, whereas "Touch NoPres-Pgs" introduces a static overhead on all transfers, which is especially bad in large, asynchronous transfers.

Pre-touching pages of small transfers with PART: The aforementioned results indicate that *it may be beneficial to touch the buffers of small transfers* (e.g. less than 64 or 256KB) *before they are RDMAed*, when the user can know that this is the first use of the buffer. In this way, we reduce the latency by as much as 3.2x in case page faults occur prior to the transfer, while increasing the latency marginally if no page-fault occurs (touching a page already in memory incurs around 100-200 nsecs of overhead).

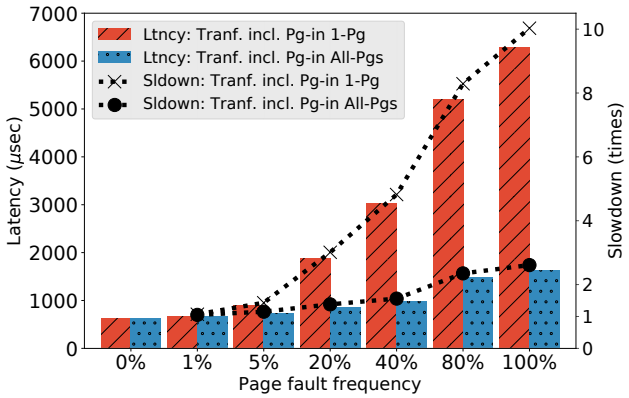

Fig. 11: Latency versus page fault frequency in 1MB transfers.

Impact of page fault frequency: In Figure 11, we evaluate the performance of PART versus the page fault frequency in 1MB transfers. The page fault frequency presents the probability that a page will exhibit a page fault, taken independently for each page. With page-fault frequency 0%, we capture the no page fault case. As can be seen in the Figure (left y-axis), for 5%, 20%, 40%, 80% and 100% page-fault frequency, the latency of PIO is 1.3, 2.2, 3.6, 5.7, and 6.7x, higher, respectively, than that of PIA.

Focusing on how PIA slowdowns the base transfer latency (0% frequency) in the presence of page-faults (right y-axis), we see that, as expected, the slowdown increases with page-fault frequency: 1.15x for up to 5%, 1.56x for 40%, and 2.6x for 100%. Thus, when the frequency of page faults is small, transfers will incur a small latency overhead.

Evaluating PART in a real HPC application: We evaluated PART by running the LAMMPS HPC application [16] on our testbed. In our experiments we compare PART, which dynamically handles all page-faults, versus a solution that touches all buffers prior to transfer, thus leading to no page faults (Transparent Huge Pages were disabled). The results depicted in Table II use four (4) threads per process—each QFDB consists of four (4) FPGAs and each one of them, running a process, has four (4) ARM A53 cores. The switches that support the connections are described in [17]–[19]. In our experiments, we varied how many processes (thus FPGAs) participated in the run, ranging from 1 up to 16 (16 FPGAs/1 blade).

As can be seen in Table II, the main principle of PART, i.e. not pinning pages and handling page-faults at the network through retransmissions, does not introduce any measurable overhead in LAMMPS. Analyzing the results, we discovered that a few only network-induced page-faults occurred at the beginning of the experiments. This happens because LAMMPS reuses its buffers that participate in communication. In PART,

TABLE II: LAMMPS performance (up to 1600 steps) using PART versus another method that pages-in all the pages of the buffers prior to the RDMA transfer.

	Processes	Loop time (sec)	Timesteps/s
No Page Faults	1	76.2046	1.312
	2	79.1183	2.528
	4	84.2753	4.746
	16	97.1399	16.471
With Page Faults	1	76.1135	1.314
	2	79.0789	2.529
	4	84.1988	4.751
	16	96.7787	16.533

the user does not have to pre-fault (touch or pin) pages prior their RDMA transfer, thus simplifying MPI programming; additionally, PART improves memory utilization, since programs can dynamically share the available memory. In future work, we want to evaluate the performance of PART with multiple programs.

VI. Conclusion

In this paper, we proposed and evaluated PART, a new end-to-end memory management scheme for RDMA. Instead of pinning pages, PART resolves occasional page faults in the network, leveraging the retransmission capabilities of modern NICs. Additionally, PART re-uses the existing IOMMU and the process page table in order to translate virtual to physical addresses, instead of deploying a separate memory management framework as modern NICs. Avoiding page pinning simplifies programming and unlocks the advantages of dynamic paging for communication buffers. We have implemented both the hardware and software components of PART in a real RDMA network connecting low-power ARM processors. Our results showed that PART *(i)* can significantly reduce the latency compared to pinning for small transfers when no page fault incurs, *(ii)* introduces small overheads for 1MB transfers (up to 1.1× higher transfer time) when a small percentage (up to 5%) of pages incur a fault and *(iii)* up to 2.6× when all pages incur a page-fault. We also evaluated a heuristic that touches buffers prior to the transfer, which can be combined with PART, especially for small, synchronous transfers. Our experiments with a real HPC application further showed that PART does not affect the application execution time.

References

[1] E. Markatos and M. Katevenis, "User-Level DMA without Operating System Kernel Modification." in *Proc. HPCA, USA*, 1997.

[2] E. V. Carrera *et al.*, "User-Level Communication in Cluster-Based Servers." in *Proc. HPCA, USA*, 2002.

[3] A. Dragojević *et al.*, "FaRM: Fast remote memory." in *USENIX NSDI*, 2014.

[4] Y. Zhu *et al.*, "Congestion Control for Large-Scale RDMA Deployments." in *Proc. ACM SIGCOMM*, 2015.

[5] A. Tavakkol *et al.*, "Enabling Efficient RDMA-based Synchronous Mirroring of Persistent Memory Transactions." *CoRR*, vol. abs/1810.09360, 2018.

[6] J. Liu, J. Wu, and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand." *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[7] C. Bell and D. Bonachea, "A New DMA Registration Strategy for Pinning-Based High Performance Networks." in *Proc. (IEEE) IPDPS, France*, 2003.

[8] H. Tezuka *et al.*, "Pin-Down Cache: A Virtual Memory Management Technique for Zero-Copy Communication." in *PIPPS/SPDP, USA*, 1998.

[9] F. P. Werner and A. Gustavo, "Minimizing the hidden cost of RDMA." in *IEEE ICDCS*, 2009.

[10] 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 TPDS*, vol. 16, no. 9, pp. 830–840, 2005.

[11] S. Novakovic *et al.*, "Storm: a fast transactional dataplane for remote data structures." in *Proc. ACM ICSS*, 2019.

[12] C. Guo *et al.*, "RDMA over Commodity Ethernet at Scale." in *Proc. ACM SIGCOMM*, 2016.

[13] I. Lesokhin *et al.*, "Page Fault Support for Network Controllers." in *Proc. ACM ASPLOS, China*, 2017.

[14] Xilinx, *Zynq UltraScale+ MPSoC Technical Reference Manual UG1085*, 1st ed., 2016.

[15] ARM, "IOMMU API for ARM architected SMMU implementations." https://elixir.bootlin.com/linux/v4.9/source/drivers/iommu/arm-smmu.c, 2013.

[16] S. Plimpton, "Fast Parallel Algorithms for Short-range Molecular Dynamics." *J. Comput. Phys.*, vol. 117, no. 1, 1995.

[17] M. Katevenis *et al.*, "The ExaNeSt Project: Interconnects, Storage, and Packaging for Exascale Systems." in *Euromicro Conference on DSD, Limassol, Cyprus*, 2016.

[18] R. Ammendola *et al.*, "The Next Generation of Exascale-Class Systems: The ExaNeSt Project." in *2017 Euromicro Conference on DSD*, 2017.

[19] M. Ploumidis *et al.*, "Software and Hardware co-design for low-power HPC platforms." in *International Conference on High Performance Computing*, 2019, pp. 88–100.