REMAP: Remote mEmory Manager for disAggregated Platforms

Dimitris Theodoropoulos
dtheodor@ics.forth.gr

Andrea Reale
REALEAN2@ie.ibm.com

Dimitris Syrivelis
Dimitris.Syrivelis@ibm.com

Maciej Bielski
m.bielski@virtualopensystems.com

Nikolaos Alachiotis
nalachio@ics.forth.gr

Dionisios Pnevmatikatos
pnevmati@ics.forth.gr

ICS / FORTH, Greece *IBM Research, Ireland* *Virtual Open Systems, France*

***Abstract*—Disaggregated computing is a new approach that promises to alleviate the problem of fixed resource proportionality in datacenter deployments. Two critical factors that affect the overall performance of disaggregated platforms are remote memory access latency and throughput. Previous works primarily expose remote data processing at the applcation level that (a) require code annotations and/or the use of custom user-level libraries, and (b) may hinder the overall system protection and functionality. In this paper, we are taking a different approach: we propose the Remote mEmory Manager for disAggregated Platforms (REMAP), a hardware architecture that enables the hotplug of remote memory resources to processing nodes, as normal paged memory at the OS-level, without requiring application-level code modifications. REMAP tightly couples processing nodes with remote memory controllers. Our architecture "expands" system memory on demand, by dynamically attaching remote memory modules to unused Local Physical Address (LPA) ranges, where the memory access requests are tunneled over high-speed, low-latency serial links. To evaluate REMAP in terms of performance, we implemented a prototype using two zcu102 FPGA boards. REMAP provides a remote cache-line access latency of less than 750 nsec, and up to 1.3x overall system throughput, compared to a baseline CPU-memory configuration.**

I. Introduction

The increased computational and memory requirements of applications from various domains (e.g. finance, astrophysics, bioinformatics, social media / networks, etc) have led researchers and companies to adopt data processing on cloud resources, and this trend is intensifying. Typically, cloud-based processing is conducted on commodity trays that integrate fixed computational and memory resources. In addition, applications (i) may use datasets that can easily reach the petabyte range, and (ii) require latency-critical results. Towards reducing response latency, application data need to be kept "close" to the processor (i.e., RAM), leading to memory footprints that can easily exceed the tray capacity.

To overcome this limitation when required, the local processor OS usually applies techniques like disk swapping, which severely affect the overall system / application performance and response time. Other systems utilize distributed memory object caching systems (e.g., memcached [1]) to logically combine memory resources over a single large virtual pool of memory. However, network latency can still severely impact the overall datacenter performance. For instance, more than 80% of Facebook's memcached latency is attributed to the network [2].

To address the increased memory space requirements of datacenter applications, disaggregated computing relies on dynamic attachment of remote memory resources to processing nodes over network links. Tray-to-tray communication is normally based on network protocols (e.g., TCP/IP), leading to excessive link latency and reduced effective throughput. It is estimated that the Network Interface Card and OS Network Stack could introduce a 2.5-32 usec and 15 usec latency respectively in TCP datacenter communication [2]. Previous works suggest bypassing the network protocol complexity by moving remote data processing at user space and perform RDMA requests [3], [4]. These approaches usually require source code annotation based on custom programming models with specific user-level libraries, potentially requiring additional software mechanisms to ensure protection between kernel and user space [5].

In this work, we propose the Remote mEmory Manager for disAggregated Platforms (REMAP), a hardware architecture that facilitates dynamic and fully transparent attachment of remote memory resources to processing nodes. REMAP is tightly coupled with processing nodes and remote memory controllers; it maps Local Physical Address (LPA) ranges to remote memory modules, and transmits requests directly over low-latency links. REMAP does not require a custom programming model or user-library, thus legacy application code can be instantly executed on larger memory spaces when available. More specifically, in this paper we make the following contributions:

- We present the REMAP architecture that allows dynamic management of remote memory regions attached to processing, over configurable link widths and remote memory segment granularity.
- We explore AXI4 transaction [6] packetization towards optimal network link utilization and efficient remote memory access.
- We describe the REMAP implementation targeting an ARM-based MPSoC with programmable logic (PL).

- We evaluate REMAP in terms of remote memory access throughput and latency, under different configurations, and compare against related work.

The paper is organized as follows. Section II discusses related work. Section III presents the REMAP architecture, Section IV describes the OS support for dynamic memory attachment, Section V describes memory transactions packetization for network transmission, Section VI describes our REMAP implementation, Section VII presents our experimental results, and Section VIII concludes this work.

II. Related work

Latency and throughput have already been identified in the research community as two primary key factors that can severely affect the overall performance of datacenters. "The Machine" project from HP Enterprise explores server-class disaggregated architectures comprising of compute and memory resource pools [7]. Similarly, the dRedBox research project, focuses on a low-power datacenter architecture that shifts toward a flexible and software-defined block-as-a-unit paradigm [8]. Towards building exascale-class machines, Exanest [9], in collaboration with ExaNode [10] and EcoScale [11] projects, study the adoption of low-cost and power-efficient ARM processor clusters. ExaNeSt nodes consist of four Zynq Ultrascale+ FPGAs, with 64 GB of local DRAM and SSD storage, connected with low-latency links, and are supported over fully-distributed NVM (Non-Volatile Memory) storage.

Montaner *et al.* describe a hardware architecture that facilitates remote attachment of memory regions to processing nodes [12]. The proposed architecture is implemented on an FPGA-based card connected to a Supermicro H8QM8-2 motherboard with four AMD quad-core Opteron processors, all communicating using the HyperTransport protocol. Moreover, to reduce node-to-node communication latency, certain works suggest bypassing the OS and network protocol complexities, by exposing remote data processing at user-space [2]. For example, [3] suggests user-level initiated RDMA, in order to bypass the kernel latency. To evaluate their proposal, the authors implemented an experimental setup connecting two Zynq Ultrascale+ MPSoCs over a Kintex Ultrascale-based switch. The Scale-Out NUMA (soNUMA) introduces an architecture and communication protocol, for reducing remote memory access latency [4].

These approaches though require annotation of the source code, using custom programming models with specific user-level libraries. In addition, I/O processing at user-level requires software mechanisms to ensure protection between user and kernel space [5], otherwise it could potentially compromise the overall system security and functionality.

III. The REMAP architecture

Remote resource management concept: Figure 1 illustrates an Application Processing Unit (APU) of a processing node connected to a 1GB local memory_0 (LPA 0x4_0000_0000 - 0x4_3FFF_FFFF). At any given instance applications require additional memory, instead of performing high-latency techniques like disk swapping, the APU OS attaches the 256MB remote memory_1 over external Master Interface 0 (MI/F_0) at LPA range 0x4_4000_0000 - 0x4_4FFF_FFFF. All memory transactions falling within this LPA range are forwarded via MI/F_0 to hardware logic, that manages and serializes them for transmission to the remote module over serial links, providing high-throughput and low-latency memory access. Based on application memory demands, the APU OS can attach an arbitrary number of *N* remote memory regions, accessible through *A* MI/Fs over a unique LPA range.

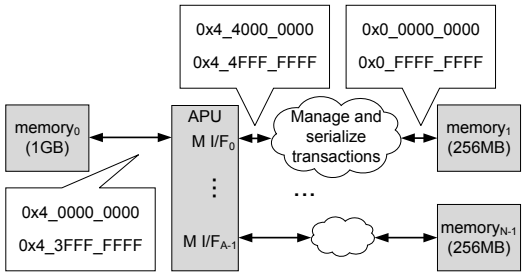

Figure 1. Remote memory attachment: APU I/O interfaces are assigned unique LPA spaces, and hardware logic manages / serializes memory transactions for transmission to the required remote location.

In order to successfully interface remote memory modules, in the egress direction (from the APU perspective) we need to take three distinct steps: (a) break memory transactions into small chunks (databeats), (b) transmit them to the correct destination resource, and (c) re-assemble databeats back into valid transactions, ready to be forwarded to the memory controller. Moreover, to send back memory responses (i.e., write acks or read data), we need to (a) again break them into databeats, (b) transmit them back to the processing node, and (c) reconstruct the original memory responses, before forwarding them to the APU.

REMAP architecture: Based on the above, REMAP introduces two distinct blocks for application execution and hosting memory resources, namely the *cBlock* and *mBlock* respectively. A cBlock tightly couples the APU with logic that (a) manages allocated remote memory regions (called segments), and (b) transmits requests / receives responses. On the other hand, an mBlock integrates logic that (a) receives memory requests, (b) forwards them to the local correct memory controller, and (c) transmits back responses.

More specifically, Figure 2 shows the cBlock internal architecture, divided into the egress and ingress pipelines exposing *C* physical links. We assume that the local APU provides *A* parallel I/O master interfaces (MI/F_0 ... $\text{MI/F}_{\text{A-1}}$). On the egress pipeline, each APU MI/F is connected to the *databeat generator logic* that integrates the *streamer* and

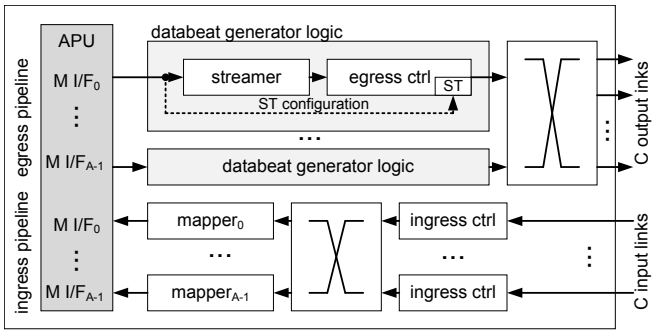

Figure 2. The cBlock architecture supporting A MI/Fs and C links: On the egress pipeline, MI/Fs transmit memory transactions, which are serialized and, based on their target address, transmitted over the correct output link. On the ingress pipeline, incoming databeats are assembled back into valid transactions and forwarded to their original MI/F.

egress controller. The former maps memory requests to a stream interface, whereas the latter breaks streamed requests into databeats to be transmitted by the local PHY over the network.

The egress controller includes also the *Segment Table* (ST). For each remotely allocated segment by the local APU, ST keeps (a) the required offset that needs to be applied to the original memory request LPA, in order to target the correct destination Physical Address (PA), and (b) the link id that connects the current cBlock with the target mBlock. For each outgoing streamed request, the egress controller (a) accesses ST to find the target remote segment, (b) converts its LPA into the correct PA, (c) breaks it into databeats, and finally (d) annotates each databeat with the correct link id, and its origin APU MI/F id. At the end, each instance of the databeat generator logic forwards link-annotated databeats to a local switch, which relays them to the correct output physical link for transmission to the destination mBlock.

On the ingress pipeline, incoming databeats from each physical link are forwarded to the *ingress controller*, which assembles them back into streamed memory responses. Each instance of the ingress controller connected to the cBlock physical links forwards all streamed responses to a local switch, which relays them to the correct *mapper*. The latter re-maps streamed responses into their original parallel representation, and forwards them to the correct APU MI/F.

Furthermore, Figure 3 shows the mBlock internal architecture with M physical links and N memory controllers. Each bidirectional physical link is connected to an instance of the *link logic*; on its ingress pipeline, it includes the *ingress controller*, which assembles incoming databeats into valid streamed memory requests. Moreover, based on their origin APU MI/F id, the ingress controller forwards each streamed request to the corresponding *mapper* instance for conversion back into its original parallel representation.

On its egress pipeline, the link logic integrates A *streamers* that send the streamed representation of memory responses to an *egress controller* instance. The latter (a) breaks

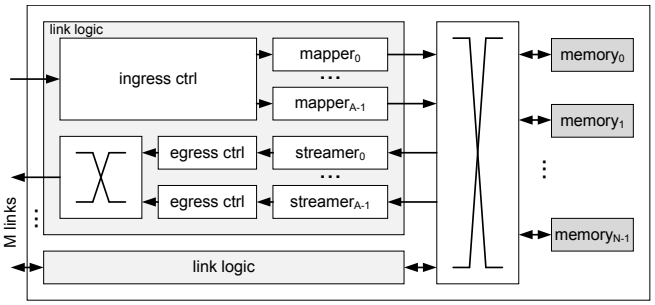

Figure 3. The mBlock architecture supporting M links and N memory modules: On each ingress link, databeats are assembled back into valid transactions and forwarded to the correct memory module. Memory responses are serialized and transmitted back over the same link.

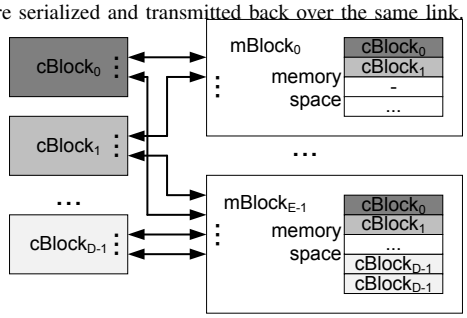

Figure 4. Interconnection of D cBlocks and E mBlocks: Each cBlock link is connected to an mBlock link, in order to access its remote memory segment.

streamed responses into databeats, and (b) annotates each databeat with the destination APU MI/F id. All annotated databeats are sent to a local output switch for transmission back to the original cBlock via the same physical link from which memory requests arrived. Finally, each bidirectional physical link is connected to an instance of the link logic, whereas a local memory interconnect provides access to all available memory modules.

REMAP imposes no restrictions on the segment allocation policy. For example, Figure 4 illustrates D cBlocks having allocated remote segments on E *mBlocks*: cBlock_0 and cBlock_1 both reserved a memory segment to mBlock_0, accessible over their first physical link, and a second segment to $\text{mBlock}_{\text{E-1}}$, accessible over their second link. Moreover, $\text{cBlock}_{\text{D-1}}$ aggregates both of its physical links to reserve segments only to $\text{mBlock}_{\text{E-1}}$, thus establishing a "wider" connection between them.

IV. OS SUPPORT

In REMAP, the OS is responsible of (i) managing attached remote memory segments and offering them transparently to user space application issuing memory allocation requests, and (ii) exposing a software configuration channel to set up corresponding OS memory management data structures and the cBlock's ST at remote memory attachment time.

We assume that access to memory by user space applications occurs via paged virtual memory: this is true for most modern general purpose operating systems. As explained in

Section III, cBlock inserts remote memory segments in the physical memory address space (as seen by the APU) by mapping it to an unused LPA (local physical address) range. In order for user-space applications to use this new memory transparently, the operating system's virtual memory management (VMM) subsystem needs become aware of new remote memory segments as they are attached, so that it can map processes virtual address space ranges on corresponding remote memory pages. To do so, whenever a remote segment is attached, the operating system's kernel alters its kernel page tables to create a virtual map that associates LPAs to kernel virtual addresses (VAs). This lets the kernel access remote memory via standard loads and stores on VAs and, therefore, applications can still allocate memory resources via standard system calls (e.g., POSIX `malloc()` [13]). The kernel can freely make use of remote memory at allocation or at swap-in time by setting user processes' page tables accordingly. Unaware of these mappings, applications keep accessing their virtual memory and the MMU takes care of translating these accesses to LPAs, which are then decoded and used by the system's cBlock.

The operating system also plays the important role of exposing a software interface allowing to control the dynamic attachment of remote memory. This interface accepts as input the LPA at which the new segment should be inserted at and the offset mapping it to the remote memory's PA (assuming that the OS has a way to learn this information). Via a device driver controlling the cBlock, a call to the interface sets up the cBlock's ST; furthermore, a call to this interface triggers the attachment of remote memory to the kernel's VMM subsystem described above. We implemented these functionalities using the Linux kernel. We reused a port of the existing memory hotplug subsystem [14] to enable dynamic rewriting of kernel page tables [15], and we expose the cBlock configuration interface via a sysfs object.

V. Transaction packetization

Target platform - system parameters: Many research projects [16], [17], [18], [8] have already proposed HPC server architectures based on ARM-based embedded processors, optionally coupled with programmable logic (PL). The majority of these MPSoCs use AXI4-based Memory Mapped (MM) interfaces [6] to efficiently couple the integrated processor with available PL resources. Following this path, in our work we consider ARM-based processor-PL configurations that use 40 bits for addresses, 16 bits for each unique AXI4 transaction id, and 128 bits field for data, applicable to a wide range of embedded platforms.

Streaming requests and responses: The AXI4 protocol supports the following transaction types: (a) write address (*wa*), read address (*ra*), write data (*wd*), read data (*rd*), and write response (*wack*). AXI4 MM transactions with 40 bits address, 128 bits data, and 16 bits id fields need to get multiplexed in time for transmission and therefore require a 147bit-wide datapath (rounding up to 19 bytes) to serve the AXI4-Stream domain [19] (referred to as "streamed transactions"). However, multi-node processing systems integrate PHYs with variable (and usually narrower) datapath widths, thus it is of paramount importance to efficiently downsize the AXI4 stream datapath for transmission over serial links.

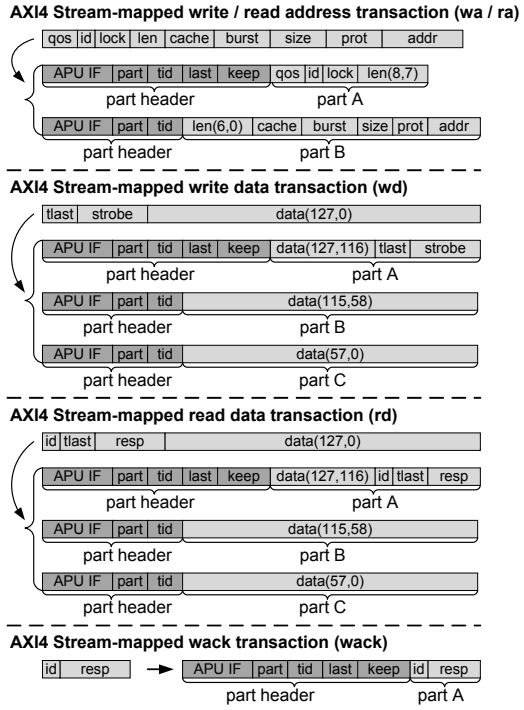

Figure 5. Breaking AXI4 streamed transactions into 64bit parts: Each part is annotated with its originating APU MI/F, ascending order (part), and the mapped transaction type (tid).

Table I
Number of required databeats when transmitting transactions pipelined and back-to-back (b2b).

transfer	burst size (b)	writes	reads
pipelined	odd	$3\frac{b+1}{2}+2$	$3\frac{b+1}{2}+1$
	even	$3\frac{b}{2}+3$	$3\frac{b}{2}+2$
b2b	odd, even	$2b+2$	$2b+1$

Having in mind as baseline 64bit PHY streams, Figure 5 shows how all AXI4 MM transaction types can be broken into 64bit databeats. According to [19] that maps transactions between the MM and Stream domains, wa / ra transactions require 81 bits, thus are broken into two 64bit databeats. wd and rd transactions require 145 and 147 bits respectively, hence are broken into three 64bit databeats. Finally, wack transactions can fit into a single 64bit part, since they require only 18 bits. REMAP annotates each part with its originating APU MI/F, ascending order (part), and the AXI4 MM transaction type (tid), in order to successfully break and re-assemble back databeats into valid AXI streamed transactions (discussed in Section VI).

When PHY streams are 128 bits, REMAP pipelines 64bit databeats of wd and rd transactions, instead of sending them

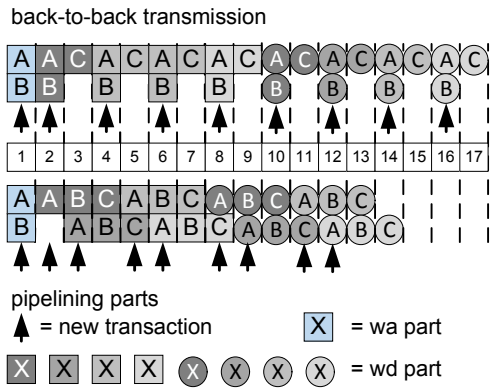

Figure 6. AXI transaction parts transmitted pipelined and back-to-back (b2b): For both cases, this example illustrates writing 128 bytes of actual payload using 1 wa and 8 wd transactions (b=8, but excluding the corresponding wack). Vertical arrows indicate a new AXI transaction for transmission. rd and wack transactions are transmitted in a similar way.

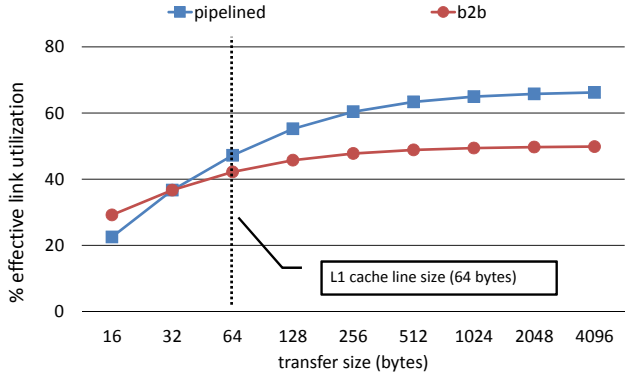

Figure 7. Effective 128bit PHY stream link utilization vs. transfer size.

"back-to-back" (b2b), leading (even for small burst sizes) to lower latency and improved link bandwidth utilization. For example, Figure 6 shows transmitting 1 wa and 8 wd transactions ($16\ bytes\ payload\ /\ wd\ \cdot\ 8\ wd\ =\ 128\ bytes\ total\ payload$) over a *128bit* PHY stream. B2b transmission requires 1 cc / wa, however needs 2 cc / wd (bits 0...127 are sent in the first cc, and 128...144 in the second cc), resulting into a total of 17 cc. On the other hand, a pipelined approach, requires 1 cc / wa and 13 cc for all 8 wds, reducing latency by $(1 - \frac{14\ cc}{17\ cc}) \cdot 100 = 17.6\%$, and increasing the *effective* link bandwidth from $\frac{128\ bytes\ payload}{17\ cc \cdot 16\ bytes/cc} = 47\%$ to $\frac{128\ bytes\ payload}{14\ cc \cdot 16\ bytes/cc} = 57.1\%$.

Table I provides the required databeats when transmitting transactions pipelined and b2b, with respect to the burst size (b = 1...256), over a 128bit PHY stream. For writes, these equations *include* also the associated wack for each wa transaction. We used the equations of Table I to calculate the expected effective link utilization.

Figure 7 projects on its y-axis the expected effective % link utilization, respectively, for both cases. As observed, when b=1 (i.e. 16 bytes), b2b and pipelined transmissions lead to an effective link utilization of 29% and 23% respectively. However, as b increases (i.e transfer size increases), pipelined transmissions become more efficient, leading up to approximately 66% effective link utilization, compared to 50% for b2b ones. Finally, one particular case is when b=4 (64 bytes = APU L1 cache line size [20]), where pipelined transmissions increase the PHY stream throughput by 12% compared to b2b ones.

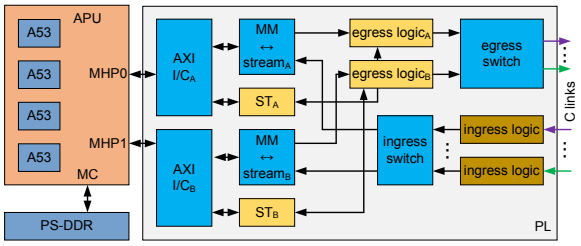

Figure 8. cBlock implementation to a Zynq Ultrascale+ MPSoC: The integrated APU provides two MI/Fs (MHP0 and MHP1) for interfacing remote memory segments over *C* links.

VI. System Implementation

cBlock implementation: Figure 8 shows the cBlock architecture implementation to a Zynq Ultrascale+ MPSoC, supporting 2 APU MI/Fs (MHP0 and MHP1) and *C* bi-directional physical links. The APU uses the PS-DDR memory to host the local OS. On the egress pipeline, its AXI4 MHP0 interface sends memory requests to the *AXI I/C$_A$* interconnect module for either configuring the *ST$_A$* or accessing a remote memory location. Memory requests are forwarded to *MM* $\leftrightarrow$ *stream$_A$* module [19], which maps them to the Stream domain, and sends them to the *egress logic$_A$* module. Similarly, the AXI4 MHP1 interface can either configure the *ST$_B$* instance or access a remote memory location, by sending memory requests to *MM* $\leftrightarrow$ *stream$_B$* to be mapped in the Stream domain, and then be forwarded to *egress logic$_B$*. Both *egress logic* instances break outgoing streamed transactions into databeats, each annotated with its originating APU MI/F, ascending order (part), mapped transaction type, and outgoing link id. All annotated databeats are forwarded to the *egress switch*, which sends them to the correct PHY IP in a Round-Robin fashion.

On the ingress pipeline, each link PHY IP sends incoming databeats to an instance of the *ingress logic* module. The latter assembles them back into valid AXI streamed responses, and annotates them with the destination APU MI/F. All annotated streamed responses are forwarded to the local *ingress switch*, which relays them back to the target *MM* $\leftrightarrow$ *stream* module for conversion back into the MM domain. All MM responses are sent back to their origin APU MI/F via the corresponding *AXI I/C* interconnect module. As observed, the cBlock implementation is modular: The egress pipeline should be instantiated *A* times in order to support equal number of MI/Fs, all connected to the *egress switch*. Moreover, on the ingress pipeline, system developers need

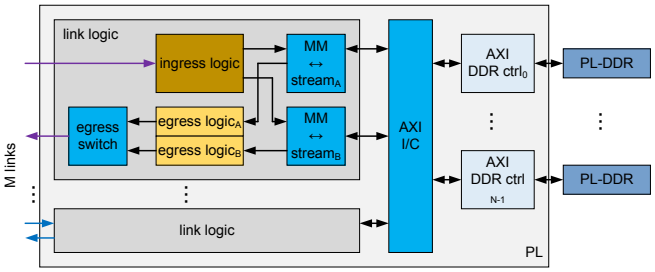

Figure 9. mBlock implementation: For each ingress link, the *link logic* module assembles databeats into streamed transactions, whereas for egress links it breaks streamed responses into databeats. An AXI I/C provides connectivity between M links and N memory controllers.

only to instantiate the *ingress logic* module C times, and connect them to the *ingress switch*.

mBlock implementation: Figure 9 shows the mBlock implementation supporting access over M links to N memory controllers. Each bi-directional link is connected to an instance of a *link logic* module. On the ingress path, the *ingress logic* accepts incoming databeats from the local PHY IP, and assembles them into valid streamed requests. Moreover, based on the annotated APU I/F id, it forwards requests to either $MM \leftrightarrow stream_A$ or $MM \leftrightarrow stream_B$ for conversion back into the MM domain. Ingress pipelines from all links can access the correct memory controller via the AXI I/C.

On the egress path of each link, the AXI I/C forwards memory responses back to the $MM \leftrightarrow stream$ module that issued the corresponding memory request. The latter converts them into the Stream domain, and forwards them to the correct *egress logic* module that breaks streamed responses into databeats and annotates them with the APU MI/F, ascending order (part), and mapped transaction type. All annotated databeats are forwarded to the *egress switch*, which relays them to the local PHY IP for transmission back to the correct cBlock in a Round Robin fashion. The mBlock implementation is completely modular: System developers need only to instantiate the *link logic* module M times, each connected to a bi-directional link. All instances can have access to N memory controllers via the *AXI I/C* module.

VII. Experimental Results

CAD tools - evaluation platforms: To implement and evaluate REMAP, we used the Xilinx Vivado 2017.1 design suite. Moreover, we generated all custom modules using the Vivado 2017.1 HLS. To run our experiments, we used two Xilinx zcu102 FPGA boards (zcu102-A and zcu102-B), each featuring a Zynq UltraScale+ MPSoC (XCZU9EG series).

Experimental configurations: Our target is to evaluate REMAP in terms of local and remote single-link performance impact (latency and throughput) for accessing memory, and to showcase the benefits of supporting dynamic attachment of remote memory modules. Figure 10 illustrates all experimental memory configurations, whereas Figure 11 shows our actual experimental setup.

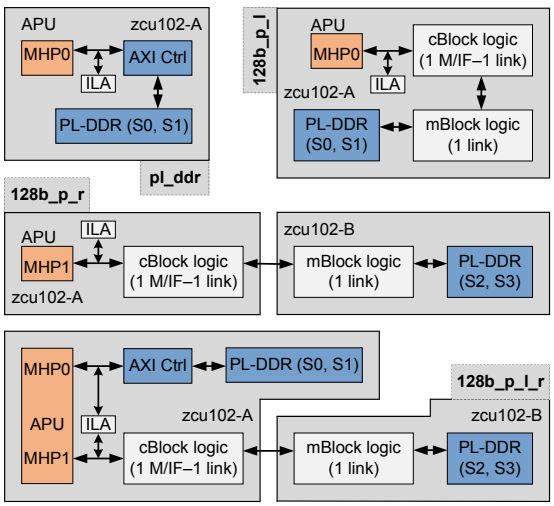

Figure 10. Configurations used in our experiments: pl_ddr is the baseline APU-memory system, 128b_p_l integrates a cBlock-mBlock single-chip implementation, 128b_p_r connects two boards over a physical link, and 128b_p_l_r combines the pl_ddr and 128b_p_r configurations in a dual-board system.

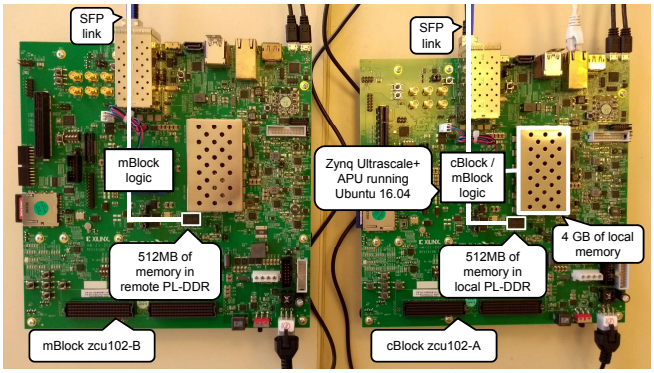

Figure 11. The prototype platform connecting zcu102-A and zcu102-B boards over an SFP-based link.

As a baseline APU-memory system, we implemented the *pl_ddr* configuration, which directly attaches the local PL DDR memory to the APU. For (a), we implemented the single-chip *128b_p_l* configuration, which uses a cBlock module with 1 MI/F and 1 bi-directional link connected to an mBlock, also with 1 bi-directional link to access its local 512MB PL DDR memory. The APU uses its MPH0 interface to access remote memory. This configuration directly evaluates only the logic performance overheads, i.e. excludes any external physical link overheads.

For (b) we implemented the *128b_p_r* configuration, which uses both FPGA boards. zcu102-A hosts a cBlock, which integrates an APU connected over MHP1 to logic supporting 1 MI/F and 1 bi-directional link for remote memory access. Moreover, zcu102-B hosts an mBlock that uses 1 bi-directional link to access its 512MB PL DDR memory. The two boards are connected with 2 SFP cables over a 20Gb link, using the Xilinx Aurora PHY IP [21]. For

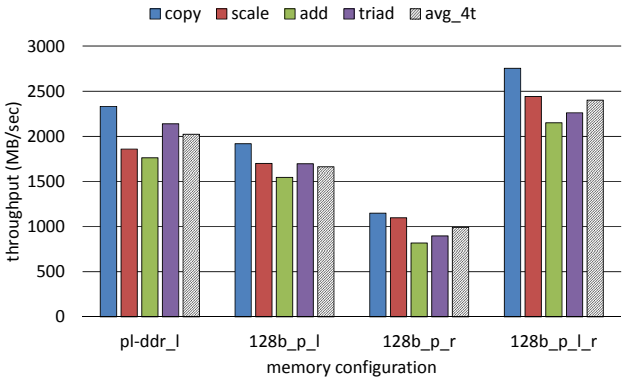

Figure 12. Stream benchmark throughput comparison among all memory configurations when using 4 threads: The 128b_p_l_r configuration provides up to 30% improved throughput compared to the baseline.

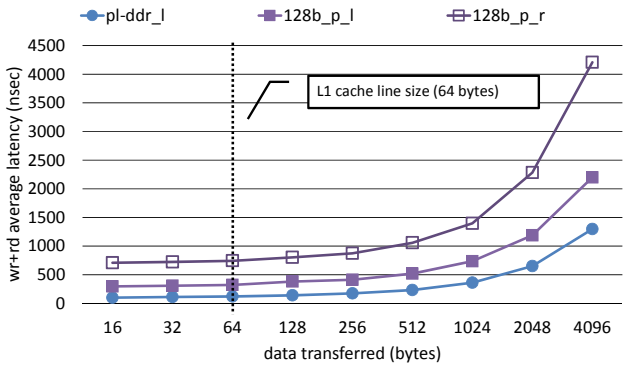

Figure 13. Average latency when accessing data to remote memory segments: Logic latency for accessing cache lines (64 bytes) can be as low as 323 nsec, whereas end-to-end latency is 748 nsec.

(c) we implemented the *128b_p_l_r* configuration, which attaches the PL DDR memory of zcu102-B to the APU of zcu102-A, accessible via its MHP1 interface over a 20Gb link. Note that each APU MI/F supports accessing up to 228 GB of remote memory.

In each memory configuration, the zcu102-A APU hosts a Petalinux OS distribution. As soon as the APU finishes booting, we first attach two 256MB segments (S0 and S1) in the zcu102-A PL DDR memory (discussed in Section IV); the OS configures ST_A so the first segment is mapped between LPA 0x4_0000_0000 - 0x4_0FFF_FFFF, with PAs between 0x0000_0000 - 0x0FFF_FFFF, whereas the second segment is between LPA 0x4_1000_0000 - 0x4_1FFF_FFFF, with PAs between 0x1000_0000 - 0x1FFF_FFFF. Both segments are accessible over the APU MHP0 interface.

Furthermore, we attach a pair of 256MB segments (S2 and S3), accessible over the APU MHP1 interface, in the zcu102-B PL DDR memory; the OS configures ST_B so the first segment is mapped between LPA 0x5_0000_0000 - 0x5_0FFF_FFFF, with a PA rage between 0x0000_0000 - 0x0FFF_FFFF, whereas the second segment is mapped between LPAs 0x5_1000_0000 - 0x5_1FFF_FFFF, with a PA range between 0x1000_0000 - 0x1FFF_FFFF.

As a summary (also shown in Figure 10):

- *pl_ddr* and *128b_p_l* can access segments S0 and S1 over MHP0 interface (total 512 MB).
- *128b_p_r* can access S2 and S3 over MHP1 (total 512 MB).
- *128b_p_l_r* can access S0, S1 over MHP0, and S2, S3 over MHP1 (total 1024 MB).

LUT and FF resource utilization is about 12% and 9% for cBlock and mBlock logic respectively, and overall BRAM occupancy is less than 27%. In other words, REMAP-enabled disaggregated systems can be even implemented using MPSoCs with limited PL resources. Moreover, the cBlock and mBlock logic uses a 3.2ns clock (313 MHz).

Throughput results: To evaluate the actual memory throughput, we used the Sustainable Memory Bandwidth in High Performance Computers (Stream) benchmark [22], compiled with -O3 optimizations and OpenMP directives that invoke 4 concurrent threads during execution. The array size is set to 10M, allocating 228 MB of memory per benchmark execution. To ensure that memory links are utilized as much as possible, we executed concurrently two Stream instances in configurations *pl_ddr*, and *128b_p_l*, allocating 456 MB in S0 and S1, and two Stream instances in *128b_p_r* allocating 456 MB in S2 and S3. Moreover, in the *128b_p_l_r* configuration, since all segments are attached, we executed concurrently four Stream instances, two allocating memory in S0 and S1, and two in S2 and S3 segments respectively.

Figure 12 the memory throughput for each of the four Stream tests (copy, scale, add, triad), and their average, always invoking 4 concurrent threads during execution. As observed, the baseline *pl_ddr* configuration reports an average of slightly above 2000 MB/sec. *128b_p_l*, which tests the cBlock and mBlock logic throughput exclusively (i.e. no link latency), suggests an average throughput reduction of 17% compared to the baseline (1663 MB/sec).

Furthermore, *128b_p_r* that includes the remote memory access over the SFP link, still reports an average of 1000 MB/sec. This drop is expected, because, as discussed in Section V, is related to OS-generated bursts for accessing APU L1 cache lines; based on an estimated average effective link utilization of 47% for accessing 64 bytes of data, we expect an effective link throughput of $(\frac{64B}{66B} \cdot 20\ Gbps) \cdot 47\% = 1139\ MB/sec$. Finally, the *128b_p_l_r* configuration, reports an aggregated average memory throughput of 2400 MB/sec (average of 19% improvement compared to *pl_ddr*) with scale reporting even up to 30% better throughput, compared to the baseline.

Latency results: Figure 13 reports the memory access latency for *pl_ddr*, *128b_p_l* and *128b_p_r* configurations when accessing up to 4 KB of data. We note that all latencies are measured on the actual hardware by placing Internal

Table II
COMPARISON TO RELATED WORK

work	latency (nsec)	notes
REMAP	323 / 750	0-hop / 1-hop latency on actual hardware
[12]	1300	1-hop latency with external FPGA card
[4]	300[1]	*128b_p_l* + 50 nsec inter-node delay = 373 nsec
[3]	2000	2-hop latency over AXI switch

[1] assumes 50 nsec inter-node latency

Logic Analyzers (ILAs) to MHP0 and MHP1 interfaces, as shown in Figure 10. The cBlock and mBlock *logic latency* (*128b_p_l*) for accessing an APU L1 cache line (64 bytes) is 323 nsec, whereas the latency for accessing remote memory in the zcu102-B (*128b_p_r*) is less than 750 nsec.

Comparison to related work: Direct comparison of different works is not always feasible, due to different technologies and setups used, however in Table II, we report latencies from other works. As discussed in Section II, [12] reported a loopback and 1-hop remote latency of 1300 nsec and 1900 nsec respectively for accessing 64 bytes of data. Based on our ILA-tracked results, the REMAP *128b_p_l* (loopback) and *128b_p_r* (i.e. 1-hop) configurations suggests a latency of 323 nsec and 750 nsec respectively. The authors of [4] report a latency of 300 nsec for accessing 64 bytes of remote data, assuming an inter-node delay of 50 nsec. A REMAP *128b_p_l* configuration with the same inter-node delay would report a similar latency of 323+50=373 nsec. However, we should note (a) that our work provides the benefit of attaching completely transparently remote memory locations vs requiring a custom programming model exposed over a user-level library, and (b) these measurements are on actual hardware. Finally, in [3], the authors state a remote access latency of 2 usec for 64 bytes between two Zynq Ultrascale+ MPSoCs, connected over an external Kintex Ultrascale-based AXI switch. Memory requests are generated at user-level over 2 hops, thus cannot be directly compared against our results.

VIII. CONCLUSIONS

The REMAP hardware architecture allows local APU OS to dynamically attach remote memory resources at kernel level, without requiring application-level changes or libraries; instead, applications can transparently use the additional memory space available. Running the Stream benchmark on a dual-board prototype, we showed that REMAP provides (a) remote cache-line access latency of less than 750 nsec, and (b) up to 1.3x overall system throughput, compared to a baseline CPU-memory configuration.

IX. ACKNOWLEDGEMENTS

This work has been supported in part by EU H2020 ICT project dRedBox, contract #687632.

REFERENCES

[1] memcached.org, "memcached: a distributed memory object caching system," https://memcached.org, [Online; accessed 11-Jan-2018].

[2] Stephen Rumble, et. al., "It's Time for Low Latency," in *Hot Topics in Operating Systems*. USENIX Association, 2011.

[3] Pantelis Xirouchakis, et. al. , "Low Latency RDMA for High-Performance Computing on ARM Platforms," in *HiPEAC ACACES*, 2017.

[4] Stanko Novakovic, et. al. , "Scale-out NUMA," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS*, 2014.

[5] Anastasios Papagiannis, et. al., *User-Space I/O for µs-level Storage Devices*. Springer International Publishing, 2016.

[6] ARM, "AMBA AXI and ACE Protocol," Arm Holdings (Arm), Tech. Rep., 2011.

[7] The next platform, "HPE Powers Up The Machine Architecture," https://www.nextplatform.com/2017/01/09/hpe-powers-machine-architecture/, January 2017.

[8] Kostas. Katrinis, et. al, "Rack-scale disaggregated cloud data centers: The dReDBox project vision," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016.

[9] Roberto Ammendola, et. al., "The Next Generation of Exascale-Class Systems: The ExaNeSt Project," in *Euromicro Conference on Digital System Design (DSD)*, August 2017.

[10] Alvise Rigo, et. al., "Paving the Way Towards a Highly Energy-Efficient and Highly Integrated Compute Node for the Exascale Revolution: The ExaNoDe Approach," in *Euromicro Conference on Digital System Design (DSD)*, Aug 2017.

[11] Iakovos Mavroidis, et.al., "ECOSCALE: Reconfigurable computing and runtime system for future exascale systems," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016.

[12] Héctor Montaner, et. al., "A new degree of freedom for memory allocation in clusters," *Cluster Computing*, vol. 15, no. 2, June 2012.

[13] A. Josey, D. Cragun, N. Stoughton, M. Brown, C. Hughes *et al.*, "The open group base specifications issue 6 ieee std 1003.1," *The IEEE and The Open Group*, vol. 20, no. 6, 2004.

[14] kernel.org, "Linux memory hotplug documentation," Online: https://www.kernel.org/doc/Documentation/memory-hotplug.txt, accessed: January 2018.

[15] A. Reale and M. Bielski, "Memory hotplug support for arm64 — complete patchset v2," Online: https://lkml.org/lkml/2017/11/23/182, accessed: January 2018.

[16] Christoforos Kachris, et. al., "The VINEYARD approach: Versatile, Integrated, Accelerator-based, Heterogeneous Data Centres," in *International Symposium on Applied Reconfigurable Computing (ARC 2016)*, 2016.

[17] Yves Durand, et.al., "Euroserver: Energy efficient node for european micro-servers," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, 2014.

[18] Nikola Rajovic, et.al., "The Mont-Blanc prototype: An Alternative Approach for HPC Systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2016.

[19] Xilinx inc., "AXI Memory Mapped to Stream Mapper v1.1," April 2017.

[20] ARM, "ARM Cortex-A53 MPCore Processor Technical Reference Manual," Arm Holdings (Arm), Tech. Rep., 2014.

[21] Xilinx inc., "Aurora 64B/66B v11.2," October 2017.

[22] John D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," University of Virginia, Tech. Rep., 1991-2007.