A Software-defined Architecture and Prototype for Disaggregated Memory Rack Scale Systems

(Preprint)

Dimitris Syrivelis*, Andrea Reale*, Kostas Katrinis*, Ilias Syrigos†, Maciej Bielski‡, Dimitris Theodoropoulos§, Dionisios N. Pnevmatikatos§ and Georgios Zervas¶
*IBM Research, Ireland
†University of Thessaly, Greece
‡Virtual Open Systems, France
§Foundation of Research and Technology Hellas, Greece
¶University College London, United Kingdom

***Abstract*—Disaggregation and rack-scale systems have the potential of drastically increasing TCO and utilization of cloud datacenters, while maintaining performance. In this paper, we present a novel rack-scale system architecture featuring software-defined remote memory disaggregation. Our hardware design and operating system extensions enable unmodified applications to dynamically attach to memory segments residing on physically remote memory pools and use such remote segments in a byte-addressable manner, as if they were local to the application. Our system features also a control plane that automates software-defined dynamic matching of compute to memory resources, as driven by datacenter workload needs.**

We prototyped our system on the commercially available Zynq Ultrascale+ MPSoC platform. To our knowledge, this is the first time a software-defined disaggregated system has been prototyped on commercial hardware and evaluated through industry standard software benchmarks. Our initial results - using benchmarks that are artificially highly adversarial in terms of memory bandwidth - show that disaggregated memory access exhibits a round-trip latency of only 134 clock cycles; and a throughput penalty of as low as 55%, relative to locally-attached memory. We also discuss estimations as to how our findings may translate to applications with pragmatically milder memory aggressiveness levels, as well as innovation avenues across the stack opened up by our work.

***Index Terms*—disaggregation, extended memory, serverless computing, pooled computing, rack scale systems, rack scale datacenters, software-defined systems, cloud datacenters, internet-scale computers.**

I. INTRODUCTION

Resource utilization is one of the key performance indicators for internet-scale datacenter and cloud providers to optimize cost of ownership. Guaranteeing consistent high utilization of resources in large datacenters is a daunting task: typical Cloud application mixes show high diversity in terms of their computing resource requirements (i.e., CPUs, memory, storage and accelerators); for example, as studies in [1] and [2] show, the distribution of per-application Memory/CPU usage ratio can be spread over three orders of magnitude. Modern Cloud implementations largely rely on virtualization and related migration techniques to improve overall datacenter utilization by partitioning and isolating resources of bare metal servers into finer-grained units. However, as virtual machines (VMs) or containers cannot span across the boundaries of a standalone server machine, the overall resource ratio remains constrained to the proportionality imposed by the server mainboard, fixed at datacenter design time. This results in a waste of CPU cores, memory and accelerators when they are asymmetrically depleted.

Fine-grained disaggregation of datacenter resources and their organization into flexible pools has the potential to radically change this landscape. And while storage is already organised in independent resource pools in datacenters today, main memory and accelerators are still statically attached to server trays. In this paper, we present a software-defined, disaggregated memory system architecture for rack scale datacenters that, in the spirit of Software-defined Networks (SDNs) [3] configurability, allows to dynamically allocate and route disaggregated memory to compute nodes from an expandable resource pool via a logically centralized software control plane. Besides describing the design of the overall system, this paper proposes a hierarchical memory controller architecture and a disaggregated memory data path that can be used to attach main memory segments directly on the memory address bus of a multiprocessor system.

The remainder of this work is organized as follows. Section II gives an overview of our disaggragated rack-scale architecture and Section III discusses the requirements for our architecture; in Section IV, we focus on the main topic of the paper, and describe the disaggregated controller that implements the memory access data path. The control plane enabling software-defined memory allocation is introduced in Section V. In Section VI, we present the prototype implementation of our architecture on Xilinx Ultrascale+ MPSoC [4] development boards; our experimental evaluation on this prototype demonstrates that, even with the hardware prototyping constraints imposed by the chosen development platform, our design can achieve largely acceptable disaggregated memory access performance as measured with the industry-standard STREAM memory benchmark [5]. We conclude our paper by reflecting our work to state-of-the art in Section VII and summarize our findings and plans for future work in Section VIII.

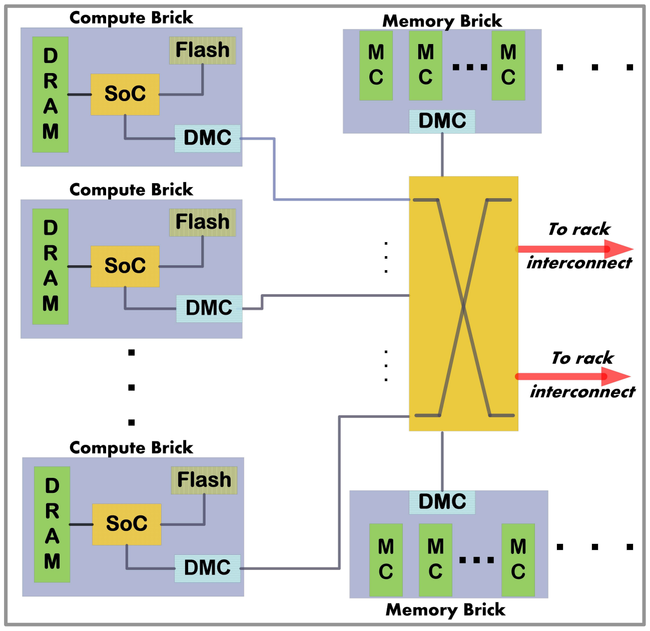

Fig. 1. Disaggregated Memory Tray Architecture.

II. System overview

The major challenge in disaggregating memory is to physically decouple it from processing elements and host them on physically and electrically autonomous components. Our disaggregated architecture proposes the physical organisation of processing and memory resources shown in Figure 1.

We refer to the physical component that hosts a multiprocessor chip as the *compute brick*. Depending on the integration approach, this component could be, for example, a System-on-Module or a pluggable card hosting a multi-processor system on-chip (MPSoC). At minimum, each compute brick also features: (a) a limited amount of local non-volatile and main memory for operating system (OS) bootstrapping and low-latency access to critical OS/user code, (b) a number of high-speed serial transceivers used to access remote and disaggregated memory, and (c) a network interface card (NIC) used to connect the brick to the control plane management network. In a production datacenter we expect a compute brick to also feature additional hardware like, for example, additional NICs for application data communication, storage interfaces, or accelerators. The compute brick runs the host OS and user applications, including VMs or containers.

Symmetrically, we refer to a *memory brick* as the physical component that hosts a number of memory controllers attaching to local memory cells (MC - e.g DDR4) as well as high-speed serial transceivers. Each memory brick controller exposes a local address space of byte-addressable memory that starts at `0x0` and grows up to the total size of memory available on the brick.

The memory brick is responsible of exposing its memory to the system's disaggregated memory pool; as we explain later, the memory brick is a passive component, meaning it has no general purpose processing capability nor configuration.

A *components tray* hosts a fixed number of standardized slots that may host any combination of compute and memory bricks. We expect trays to be packaged in datacenter rack mainboards and chassis. Serial transceivers on the bricks can be flexibly bridged with an appropriate number of on and off -tray switching layers. Trays are grouped in racks, and a full system can be made of several interconnected racks. Overall, the entire packaging and tray- and rack-level interconnection is beyond the scope of this paper; still, the work presented herein assumes that all component transceivers are interconnected via software-controlled *circuit switches*, per the high-level architecture outlined in [6]. Optical circuit switches are a more attractive solution for memory disaggregation because of their ps-level switching delay. Good candidate approaches are discussed in [7] and [8].

Finally, a management network connects compute bricks to the control plane of the system. As resource allocations change, the control plane — implemented, for example, as a set of traditional "out-of-band" server nodes arranged in hot replicas — has the role of dynamically configuring (a) circuits between compute and memory bricks and (b) compute bricks' hardware and software to establish memory access data paths. At steady state, and depending on the workloads hosted on the datacenter, the control plane allocates portions of memory residing on memory bricks (called memory segments) to compute bricks.

Transparency to workloads is the main driving and differentiating requirement of our design. Of great significance to the value of the present work is that, **from the point of view of any application running on a compute brick, allocated segments of disaggregated memory are usable as if they were local to the brick**. More precisely, once allocated, disaggregated memory segments are presented to the brick as byte addressable pieces of main memory, directly mapped into its physical address space and transparently accessible via standard memory bus transactions. The OS exposes remote memory segments grouped in distinct NUMA domains, reflecting the actual distance of the compute brick from the memory segments; this allows applications to make transparent yet qualified use of disaggregated memory.

The Disaggregated Memory Controller (DMC), the core of our contribution, is a hardware component instantiated on both compute and memory bricks in slightly different variants. It is responsible for realising the disaggregation abstraction and to make remote memory segments appear as local to compute brick CPUs. We call the DMC instantiated on compute bricks and memory bricks Compute DMC (CDMC) and Memory DMC (MDMC), respectively.

III. Requirements

A CDMC sits on the compute brick memory bus, intercepts disaggregated memory requests and routes them towards the appropriate destination. Without any loss of generality, we assume that a CDMC listens to one ore more bus master ports, i.e., entry points which can concurrently initiate memory bus transactions. Symmetrically, a MDMC sits on the memory brick's memory bus, receives incoming remote memory requests from the memory brick transceivers and routes them to the local bus slave ports, i.e., entry points able to concurrently respond to bus transactions.

In order to achieve the overall transparency goal outlined in the previous section, the CDMC needs to provide following high-level functionalities:

1) intercept disaggregated memory access requests originated by any of the master ports;
2) translate disaggregated memory transactions so that they can be understood by receiving memory bricks;
3) steer memory requests towards the appropriate serialization and de-serialization (serDES) pipelines, which will transport them toward the destination memory bricks;
4) accept incoming remote memory responses and deliver them to the master port which issued the request.

Conversely, a MDMC is a relatively simpler component that needs to:

1) accept memory requests that arrive on the memory brick transceivers and directly deliver them to a local memory controller slave port;
2) accept memory responses from the slaves and route them via the originating transceiver line.

Our design assumes a software controlled circuit-switched network forming point-to-point links between connected compute and memory brick transceivers. This guarantees that, once a circuit connecting a compute brick transceiver and a memory brick transceiver has been established, it is sufficient for the CDMC to chose the appropriate Tx transceiver link to ensure memory transaction delivery at the correct destination; in other words, we do not need to add any explicit addressing information to memory requests (responses) in order to identify the destination brick.

Given that performance is the main barrier to memory disaggregation, our DMC design needs to be latency and bandwidth sensitive. Spatial and temporal parallelism is carefully exploited throughout all aspects of its architecture. Our assumption of lossless point-to-point channels lifts the requirement for network-level acknowledgement schemes and back pressure support. Still, edge buffering techniques need to be employed to deal with possible performance asymmetry between communicating entities.

In order to not impede the deployment scalability potential, the aggregated bandwidth of the total number of transceivers on a memory brick may exceed the aggregated performance of its memory controllers. To avoid the disastrous scenario where multiple compute bricks generate a workload that the target memory brick cannot handle, the CDMC should feature rate limiting logic behind each compute brick's transceiver, globally governed by the control plane.

IV. DMC Hardware Data path

Figure 2 shows the internal architecture of the CDMC, featuring in the example 2 master and 2 transceiver ports for the sake of presentation ease. In the figure, the Tx pipelines (from the compute brick memory bus to its transceivers) are

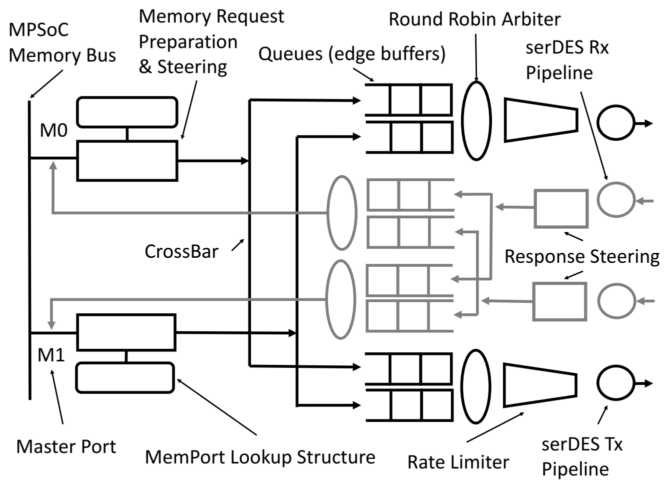

Fig. 2. Disaggregated Controller Design on the Compute Brick side

highlighted in black, while the reverse Rx pipelines are drawn in a lighter gray.

The core of the CDMC task is to intercept disaggregated memory requests and translate them in a form that is consumable by the destination memory brick's memory controllers. At remote memory segment allocation time, each remote memory segment is linearly mapped to a portion of the receiving compute brick physical address space. This mapping is arbitrary and determined by the control plane. This creates two alternative "physical" mappings for each allocated memory segment: the first (static) from the point of view of the memory brick hosting the segment, the second from the point of view of the compute brick to which it is allocated. For example, a 512 MiB memory segment that is mapped in the space `[0x20000000-0x3FFFFFFF]` from the perspective of its host memory brick might be assigned to the physical map `[0x800000000-0x81FFFFFFF]` from the perspective of the compute brick to which it is allocated.

Given this mapping scheme, it is easy to see that the CDMC translation process consists of offsetting of a physical memory address produced by the compute brick CPUs. This task is performed by the *MemPort Lookup Structure* and *Memory Request Preparation and Steering* components, instantiated once per master port (thereby shown twice in the example in Figure 2).

As depicted in Figure 3, the *MemPort Lookup Structure* stores information about currently allocated remote memory segments; it is indexed by segment address boundaries (as seen by the compute brick), and it contains all the information needed to translate and steer a request to its destination memory brick, i.e., offsetting information and identifier of the point-to-point link to the memory brick.

When a memory transaction request is received, the *Memory Request Preparation and Steering* component modifies the request using information from its associated *MemPort Lookup Structure*; before forwarding it to the proper output transceiver, it tags requests with an identifier of the master port that originated them. This identifier is eventually used on the Rx path to deliver correctly incoming responses.

The *Memory Request Preparation and Steering* component features a $1xN$ crossbar, where N is the number of available transceivers, that steers each request flit towards a destination transceiver's transmission queue. Each transceiver is driven by a $Mx1$ switch, where M is the number of master ports. The transceiver switches are input queued and just feature a Round-Robin (RR) arbiter without any crossbar because the output is single. The arbiter is designed to pull all the flits that belong to a memory transaction request before it switches to the next queue.

A similar but reversed architecture is used to steer remote memory responses back to their originating master ports. Using the tag identifying the originating master, copied from the request at the memory brick side, the crossbars behind the receiving transceivers redirect the arriving flits to the appropriate master port input queues where the same type of RR arbiter is used to complete delivery.

The described memory data path architecture features back pressure support both on the compute brick and memory brick pipelines, so, if a local queue buffer is full, the whole pipeline that pushes data to this buffer will stall. Nevertheless, back pressure support is not available between transceivers so a possible congestion at the memory brick side can result in data overflow. However, this can occur only if the aggregated bandwidth of all transceivers on the memory brick is higher than the capacity of local memory controllers. While an obvious solution is for the memory bricks to feature an appropriate transceiver aggregated bandwidth so that the memory controllers will be never overwhelmed, confining the number of brick transceivers can impede the scalability potential. Instead, the CDMC features a rate limiter component before each transceiver Tx pipeline. The limiter is configured by the control plane which, having a global view of memory allocations, can make sure that no memory brick gets a higher request volume than its controllers can handle.

The presented overall DMC design can be easily scaled to support an increasing number of on-brick CPUs and memory bricks by increasing the number of master ports and transceivers per brick. In terms of performance, the proposed architecture backplane throughput scales linearly with the number of masters within a single clock domain. All the RR arbiters in the design have a single output to feed so they do not need to run at a faster clock rate to keep the transceiver serDES pipelines fully occupied. Since it is safe to assume that the memory bus in any design will be capable of operating at significantly faster clock rates than any transceiver, the CDMC can be also partitioned in two different clock domains, one operating at the memory bus clock and one at the transceiver clock, with the edge buffering queues acting as clock domain barriers, so the backplane performance can be further improved. Our DMC design needs MxN queues on the Tx path and the same number of queues on the Rx path — M being the number of bus masters, N the number of transceivers. Evidently, this demand on hardware FIFO queues has the

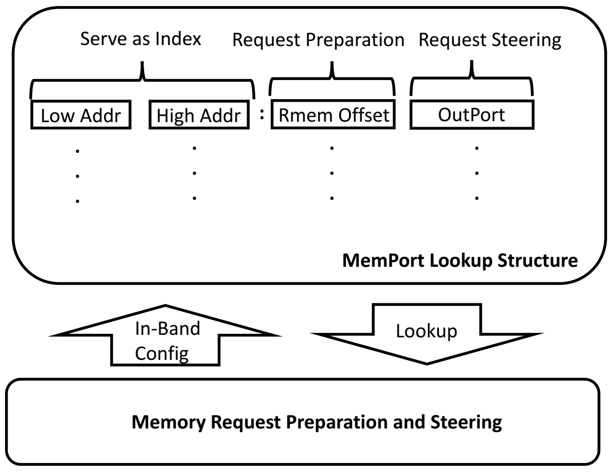

Fig. 3. Memport Lookup Structure and Memory Preparation and Steering components.

potential to limit scalability as the numbers of transceivers and masters grow. Nevertheless, hundreds of transceivers can be facilitated before this becomes a real problem. Note that the scalability analysis above shares challenges with Virtual Output Queing (VOQ) switching architectures [9], [10], [11].

V. Control plane

The control plane is responsible of maintaining an up-to-date view of the global system resource allocation state, handle incoming resource allocation requests, and configure the system hardware to accommodate these allocations.

The control plane is implemented in software and it is realized as two hierarchical components: a logically centralized *platform synthesizer* and distributed *compute brick agents*. *Compute brick agents* run on compute bricks as kernel- and user- space components. In kernel-space, a compute brick agent consists of a device driver able to push new configurations to MemPort Lookup Structures and transceivers rate limiters, and of a memory driver, built on top of Linux memory hotplug [12], which exposes newly allocated memory segments to the rest of the OS. Full description of OS-level support for our system is beyond the scope of this paper. In user-space, a networked daemon will listen to memory allocation/deallocation requests from the platform synthesizer and interact with the kernel-space components to satisfy them.

The *platform synthesizer* holds an up-to-date view of the system resources state and allocation. It provides high level interfaces to create, remove or change memory allocations, exposed as REST APIs; it acts on these requests by configuring compute bricks and by setting up the point-to-point circuits between brick transceivers via the configuration of the appropriate network switches.

System resources are modelled as a distributed directed graph, which is traversed at allocation requests to check on resource availability and to generate system interconnect configurations. Figure 4 shows an example of such a resource

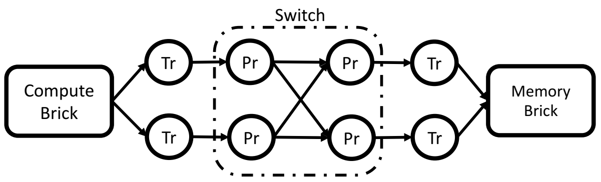

Fig. 4. An example graph-based data model for the representation of resources and reservations in the platform synthesizer

graph. Compute and memory bricks act as root and sink vertices, respectively; edges depicted as solid arrows represent the physical interconnect among system components; these edges are built to reflect all the possible configurations supported by the system hardware configuration. Evidently, a complete traversal from source to sink of this type of edges represents a point-to-point circuit connecting a compute brick to a memory brick.

The example in Figure 4 features one compute and one memory brick; each of them has 2 transceivers, all connected with a $2x2$ optical circuit switch. During the allocation stage, a path has to be selected. This path is subsequently traversed and all the visited transceiver and port vertices are marked as reserved so that all the paths that include them will not appear as options in future allocation requests. Moreover, the required resource configuration is determined during the path traversal and thus, Software-Defined memory rules are generated.

In addition to reflecting resource availability, graph vertices can be annotated with additional information describing the resource they represent including, for example, current load or power requirements. All these fields can be taken into account by graph traversing resource allocation algorithms to optimize system usage. Investigating specific resource allocation algorithms is beyond the scope of this paper.

VI. Implementation and Evaluation

We have designed and implemented a fully functional prototype of the proposed software-defined disaggregated memory architecture on the Xilinx Ultrascale+ MPSoC platform [4]. The unique feature of this platform is the integration, in a single SoC, of a so-called Processing System (PS) consisting of four ARMv8 A53 cores, and a Programmable Logic (PL) featuring a programmable FPGA. Each PS core features separate instruction and data caches of 32 KiB, and they share 1 MiB of last level cache. The cache line size is 64 bytes. Moreover, the PL is interfaced to high speed GTH serial transceivers. The memory bus interconnect is based on the AXI4 memory mapped bus architecture [13] and features two master ports towards the PL, each one of them serving a $224GB$ memory address space.

The experimental prototype is comprised of two Trenz UltraSOM+ development boards [4] featuring the XCZU9EG Ultrascale+ MPSoC. Each board has 2 GiB of DDR4, driven by two hardware DDR controllers, and 16 GTH transceivers.

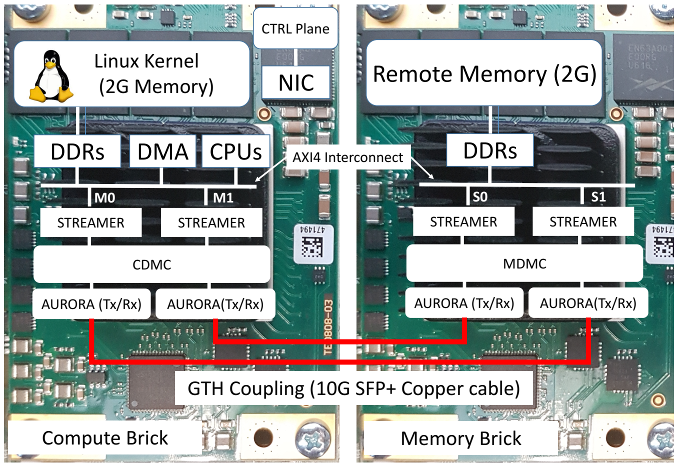

Fig. 5. Experimental prototype setup

Two of the latter are interfaced to SFP+ slots. We implemented the proposed DMC on this development platform using the protocol design approach on the Vivado High Level Synthesis [14] toolchain.

In our experiments, one of the two boards acts as compute brick, the other as memory brick, as shown in Figure 5. The PL on the compute brick receives AXI4 memory bus requests from the brick's CPUs and DMAs. A streamer component multiplexes the AXI4 bus channels in time. The width of the datapath is 152 bits, corresponding to the width of the widest AXI4 channel, i.e., the Read Data channel (128 bits data + control signals).

The CDMC receives AXI4 channel data beats, performs the translation and steering operations described in Section IV, and downsizes the data path width to 64 bits to deliver requests to the serDES pipelines. The latter are implemented using Xilinx Aurora IP cores in streaming mode with 64/66B encoder.

The memory brick receives the requests via the corresponding Aurora cores. The MDMC implements edge buffering and upsizes requests back to 152 bits before feeding them to the streamer. Finally, requests are delivered to the hardware DDRs on the memory brick via AXI4 slaves. Responses are delivered back to the compute brick via a similar but reversed pipeline.

The transceivers are clocked at 156.25 Mhz which, for a 64 bits data path width, provides a rate of 10.3125 Gbit/s per lane. The rest of the PL design runs in the same clock domain. **The critical path of our design is** 134 **clock cycles long, applying to an entire flit round-trip.** The largest contributors to this value are the Aurora pipelines, with a total of 57 cycles per direction. Note that our critical path calculations do not include signal propagation delays on cables and DDR response time, the latter being exactly the same as for local memory access. According to our measurements, our experimental prototype achieves a flit round trip delay below 1µs.

On the software side, the compute brick runs arm64 Linux kernel version 4.6.0 and Ubuntu arm64 distribution. At startup, the kernel will only see the local 2 GiB memory. We configure the CDMC via the control plane to map 4 additional 512 MiB

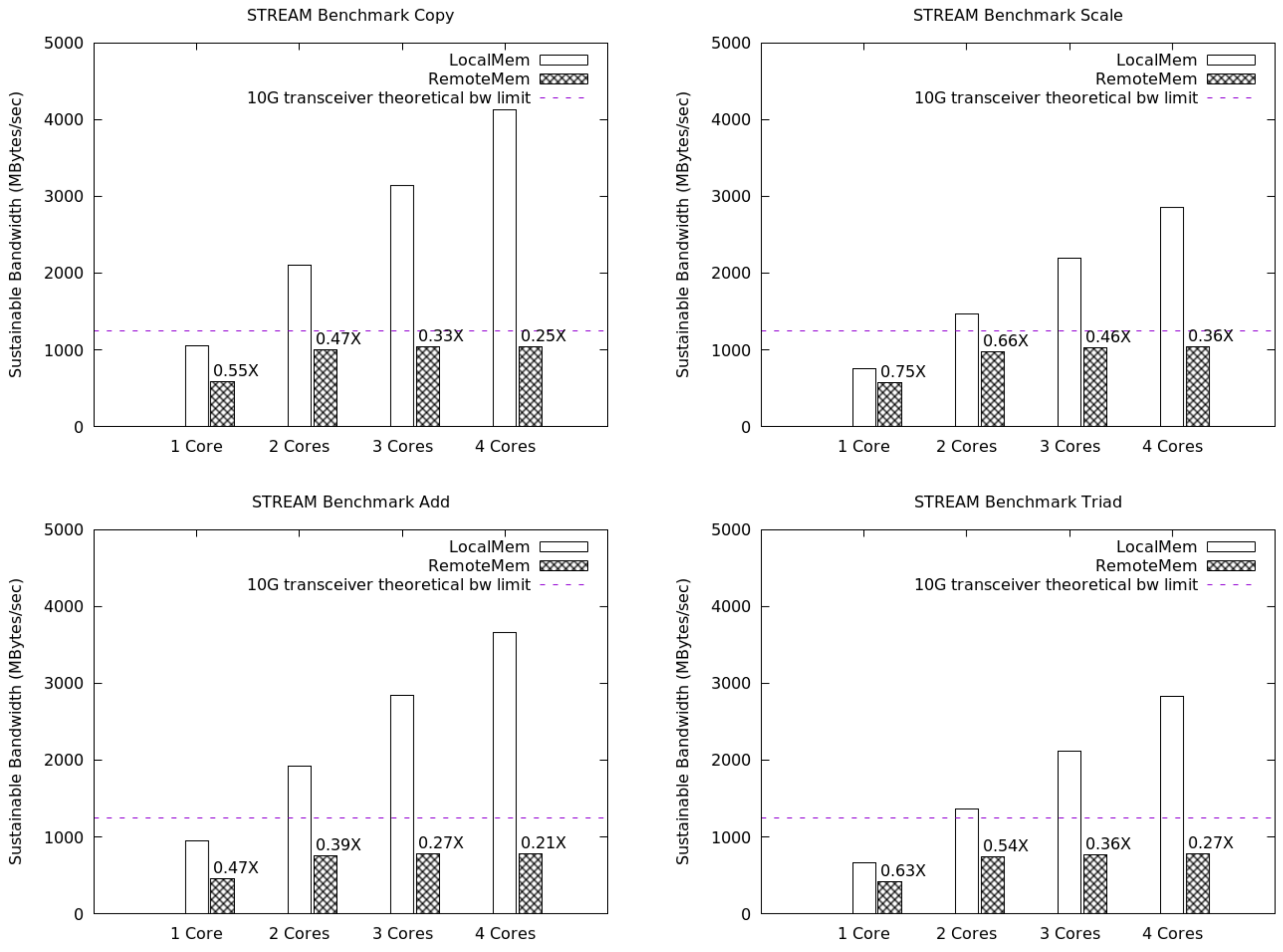

Fig. 6. STREAM benchmark performance comparison: local Vs software-defined remote memory

segments, all served by the same master port and connected to the memory brick by one of the transceivers. Leveraging our custom hotplug-based memory driver, we map this additional 2 GiB of disaggregated memory to a second NUMA domain.

Using this setup, we evaluate disaggregated remote memory performance using the STREAM benchmark [5], the de facto industry standard to measure sustainable memory bandwidth and overall processing balance as perceived by user space applications. We configured STREAM to use 10 million array elements, requiring a total memory of 228.9 MiB, which is well beyond the system cache size. Each benchmark run executes four kernels, i.e., "copy", "scale", "sum" and "triad" [5]. Specifically, "copy" reads/writes 16 bytes (1 read, 1 write ops) of memory per iteration, performing no floating point operations (FLOPs); "scale" reads/writes the same amount of memory with the same number of operations but it performs 1 FLOP per iteration; "sum" accesses 24 bytes of memory (2 read and 1 write ops) and executes 1 FLOP per iteration; finally, "triad" accesses 24 bytes of memory (2 read and 1 write ops) executing 2 FLOPs per iteration. Using the OpenMP support built-in on STREAM, we confine the benchmarks to run on 1 up to all 4 compute brick cores[1]. Leveraging the local and remote NUMA domains, we repeat the same executions using only local or disaggregated memory.

Figure 6 shows the results of our evaluation, comparing local and remote memory performance through clustered bars. The dotted line designates the maximum theoretical bandwidth that can be achieved by a 10G transceiver, i.e., 1280 MiB/s and the bars the sustainable memory bandwidth as measured by STREAM benchmark for a different number of cores.

Focusing on the "copy" kernel, the results show that one CPU core can achieve 582 MiB/s bandwidth towards remote memory, with a penalty of 45% compared to local access. As more CPUs are used concurrently, the transceiver bandwidth is quickly saturated and, beyond 2 CPUs, it becomes the performance bottleneck. In terms of absolute bandwidth, the "scale" benchmark has worst results because of the presence of the additional FLOP. However, when comparing local vs. disaggregated memory, the application-perceived penalty of using remote rather than local memory is reduced to 25%, due to the more balanced mix of memory access / processing

[1]In multicore configurations, cores operate independently on separate data.

operations. The same trend can be observed in the “add” (24 bytes memory accesses per iteration, with 1 FLOP) and “triad” (24 bytes memory accesses per iteration, with 2 FLOPs). Overall, these results validate the balanced and pipelined DMC design and implementation, showing it is capable of exploiting the full potential of the AXI4 interconnect parallel and asynchronous operation.

The experimental data collected so far show that remote memory performance can cause up to 5x deterioration of application-perceived memory bandwidth in the worst case scenario where: i) application memory access patterns cannot take advantage of processor caches, ii) 4 CPU cores run memory-heavy workloads concurrently and iii) only one master port and one transceiver are used to serve all memory access requests.

At the same time, the benchmarks also show that the introduction of just one single FLOP interleaved with memory accesses can have surprisingly positive effects on the application-perceived penalties of using remote memory. In production scenarios, we expect that delays introduced by memory disaggregation would have a much smaller impact on real world applications: unlike a memory benchmark like STREAM, these workloads would feature a much more balanced mix of memory access, floating point and I/O operations, and present a much cache-friendlier behaviour. Finally, we expect that remote memory access delay can be further and significantly brought down in a production implementation of our prototype: in fact, it needs to be remarked that our prototype DMC works at the frequency of 156.25 Mhz due to requirements imposed by the 10 Gb/s transceivers.

VII. Related Art

Making native rack scale resource pooling a reality in the datacenter through disaggregation has been a year-long quest. Results [15] obtained lately through evaluation of an analytics workload on SparkSQL have shown that – throughput-wise – memory disaggregation can be feasible even with conventional 40Gbps interconnects. Breaking the monolithic design of datacenters (including memory) to decouple arbitrary workload sizes from static server configuration and enabling component-independent technology refreshes has been one of the missions of the Open Compute Project [16] since its early days. Notable demonstrators and prototype concepts include Intel Rack Scale Design [17], Facebook "Group Hug" and "Yosemite" server designs, as well as production-grade specialized kernels and platform orchestration software for virtual machines operating on pooled servers, such as Liqid [18]. Similarly, the HPE Machine [19] prototype showcases SoCs accessing remote memory via specialized bridging controllers and fabric. Our work shares common objectives with and can act complementary to such and related designs; our unique ambition and the main differentiation point of our proposal stands in its ability to offer disaggregated memory access dynamically and transparently to unmodified application binaries and at memory-scale performance levels.

Previous work on enabling dynamic scale-up to remote memory resources has shown the potential and challenges of the approach, either transparently to consuming applications (via OS/microarchitecture cooperative disaggregation [20]), or exposing remote memory via explicit programming models [21]. These previous efforts have evaluated the potential of memory disaggregation using simulation. Our work evaluates a similar approach on real hardware showcasing a prototype based on a commercial SoC, aiming at appreciating the production potential of the approach to benchmarks and representative applications. To that end, we aim to extend our approach to incorporate targeted OS-level innovation to further improve application-perceived memory performance and expand to further applications to directly compare our findings to [22].

Shared memory clusters define a machine organization taxonomy that shares technical challenges and some of the technical and business objectives of disaggregated datacenters. Distributed shared-memory machines representative of this taxonomy, like NumaScale [23] or SGI UV [24], have emerged with the goal of satisfying parallel and distributed applications (e.g., large-scale computational science, mainframe computations) that require deployment on a large number of tightly cooperating cores, whereby also in-memory computing can bring substantial benefit. Therefore, and to cater for efficient cooperation between arbitrary sizings of cores, these machines employ cache coherency between underlying coherency domains, i.e., a distributed cache coherency protocol is typically employed to turn underlying island domains into a single cache coherent shared memory machine. At the cost of this flexibility comes scalability: distributed cache coherence protocols are not trivial to scale [25] and often impose architectural decisions to control complexity and cache coherency overhead (e.g., static organization of cooperating multi-processor in tori topologies). In contrast, our work targets typical Cloud datacenter workloads where applications are typically developed to leverage socket-level parallelism and to make heavy use of in-memory computing, but without tangible benefits from a distributed shared memory scheme. For instance, this is the case with in-memory data grids like Redis [26] that are widely used today to hide access latencies to slower stores. While arbitrary extensions of memory through disaggregation brings obvious benefits to this class of workloads in terms of utilization and performance, distributed cache coherence can be hardly of any value.

VIII. Conclusions and future work

In this paper, we have presented a software-defined architecture for memory disaggregation in warehouse-scale computer and cloud datacenters. It features a logically centralized control plane, dynamically governing a set of disaggregated compute and memory resource pools materialized as standardized bricks plugged into rack trays. Our main contribution with this work is the proposal of a novel disaggregated memory controller (DMC) hardware architecture, which allows to arbitrarily allocate and access memory segments to compute nodes according to workload-driven requirements. Uniquely to our solution, access to disaggregated memory is completely transparent from the perspective of CPUs — meaning that our system is able to run unmodified application binaries over disaggregated memory.

We have presented a prototype implementation of our design based on the commercially available Xilinx UltraScale+ MPSoC. Our initial experimental evaluation on this prototype shows that our cut-through design introduces an overhead as little as a 134 clock cycles compared to local memory access. Using the standard STREAM memory benchmark over the disaggregated infrastructure, we show that, in our prototype, remote memory exhibits a throughput penalty ranging from 1.8x to 5x compared to local memory access, when stressed against extremely memory-heavy workloads. On the far end of this range and even if real workloads are not as memory-intensive as the STREAM benchmark, we contend that this finding is an artefact of the hard constraints imposed by the hardware prototyping platform. Specificaly, we identify the master port aggregate throughput and of the 10Gb/s transceiver as the main bottleneck on our testbed; we discussed how our design can be seamlessly scaled to overcome these limitations on specialized hardware built for the purpose, by e.g. using parallel transceivers and memory bus master ports.

Our findings demonstrate that disaggregation is a viable option for next generation datacenters. In fact, compared to our development prototype, we expect production and at-scale implementations of our design to drastically bring down remote memory access overhead, for example, by trivially increasing the number and capacity of transceiver or by implementing our DMC design on ASIC, thus increasing its operating clock frequency.

Our future work includes further evaluation on representative Cloud workloads: our expectation is that the balanced mix of memory access, computation, and I/O operations, together with a more efficient use of processor caches, can significantly mitigate the penalty of using remote memory. Furthermore, we will extensively explore OS-level optimization techniques as a promising strategy to improve application performance.

Acknowledgements

This work has been conducted in the scope of the dReDBox (disaggregated Recursive Datacenter-in-a-Box) project [27]. The dReDBox project is funded by the European Union Horizon 2020 Research and Innovation programme under grant agreement No 687632.

References

[1] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *ACM Symposium on Cloud Computing (SoCC)*, San Jose, CA, USA, Oct. 2012.

[2] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker, "Network support for resource disaggregation in next-generation datacenters," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, ser. HotNets-XII. New York, NY, USA: ACM, 2013, pp. 10:1–10:7.

[3] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and openflow: From concept to implementation," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014.

[4] "Trenz ultrasom+ te0808-04," Trenz Electronic. [Online]. Available: 'https://shop.trenz-electronic.de/en/TE0808-04-09-S-TE0808-04-09-S-Starter-Kit?c=329b'

[5] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[6] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends, "Rack-scale disaggregated cloud data centers: The dredbox project vision," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 690–695.

[7] L. Schares, B. G. Lee, F. Checconi, R. Budd, A. Rylyakov, N. Dupuis, F. Petrini, C. L. Schow, P. Fuentes, O. Mattes, and C. Minkenberg, "A throughput-optimized optical network for data-intensive computing," *IEEE Micro*, vol. 34, no. 5, pp. 52–63, Sept 2014.

[8] G. S. Zervas, F. Jiang, Q. Chen, V. Mishra, H. Yuan, K. Katrinis, D. Syrivelis, A. Reale, D. Pnevmatikatos, M. Enrico, and N. Parsons, "Disaggregated compute, memory and network systems: A new era for optical data centre architectures," in *Optical Fiber Communication Conference*. Optical Society of America, 2017, p. W3D.4.

[9] Y. Tamir and G. L. Frazier, "High-performance multi-queue buffers for vlsi communications switches," *SIGARCH Comput. Archit. News*, vol. 16, no. 2, pp. 343–354, May 1988.

[10] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, "High-speed switch scheduling for local-area networks," *ACM Trans. Comput. Syst.*, vol. 11, no. 4, pp. 319–352, Nov. 1993.

[11] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, "Matching output queueing with a combined input/output-queued switch," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1030–1039, 1999.

[12] "Linux memory hotplug documentation," kernel.org. [Online]. Available: https://www.kernel.org/doc/Documentation/memory-hotplug.txt

[13] *AXI Reference Guide v13.1 UG761*, Xilinx, March 2011.

[14] K. Karras and J. Hrica, *Designing Protocol Processing Systems with Vivado High-Level Synthesis v1.0.1 XAPP1209*, Xilinx, August 2014.

[15] P. S. Rao and G. Porter, "Is memory disaggregation feasible?: A case study with spark sql," in *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, ser. ANCS 2016. New York, NY, USA: ACM, 2016, pp. 75–80. [Online]. Available: http://doi.acm.org/10.1145/2881025.2881030

[16] "Open compute project, ocp summit iv: Breaking up the monolith." [Online]. Available: http://www.opencompute.org/blog/ocp-summit-iv-breaking-up-the-monolith/

[17] "Intel rack scale design." [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design/rsd-vision-brochure.html

[18] "Liqid hyperkernel." [Online]. Available: https://liqid.com

[19] "The next platform, "hpe powers up the machine architecture"." [Online]. Available: https://www.nextplatform.com/2017/01/09/hpe-powers-machine-architecture/

[20] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 267–278, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1555815.1555789

[21] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out numa," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 3–18, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2654822.2541965

[22] H. Montaner, F. Silla, H. Fröning, and J. Duato, "A new degree of freedom for memory allocation in clusters," *Cluster Computing*, vol. 15, no. 2, pp. 101–123, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10586-010-0150-7

[23] "Numaconnect: A high level technical overview of the numaconnect technology and products (numascale whitepaper)." [Online]. Available: 'https://www.numascale.com/numa_pdfs/numaconnect-white-paper.pdf/'

[24] "Sgi uv - the world most powerful in-memory supercomputers." [Online]. Available: http://www.sgi.com/products/servers/uv/index.html

[25] M. A. Heinrich, "The performance and scalability of distributed shared-memory cache coherence protocols," Ph.D. dissertation, Stanford, CA, USA, 1999.

[26] "Redis." [Online]. Available: https://redis.io

[27] “dredbox h2020 project website.” [Online]. Available: https://www.dredbox.eu