ThymesisFlow: A Software-Defined, HW/SW co-Designed Interconnect Stack for Rack-Scale Memory Disaggregation

Christian Pinto*, Dimitris Syrivelis*, Michele Gazzetti*, Panos Koutsovasilis*
Andrea Reale*, Kostas Katrinis*, H. Peter Hofstee†

*IBM Research Europe, Dublin, Ireland
Email:{christian.pinto, dimitris.syrivelis, michele.gazzetti1, koutsovasilis.panagiotis1}@ibm.com
{realean2, katrinisk}@ie.ibm.com
†IBM Systems, Austin, TX, United States Email: hofstee@us.ibm.com

***Abstract*—With cloud providers constantly seeking the best infrastructure trade-off between performance delivered to customers and overall energy/utilization efficiency of their data-centres, hardware disaggregation comes in as a new paradigm for dynamically adapting the data-centre infrastructure to the characteristics of the running workloads. Such an adaptation enables an unprecedented level of efficiency both from the standpoint of energy and the utilization of system resources. In this paper, we present – *ThymesisFlow* – the first, to our knowledge, full-stack prototype of the holy-grail of disaggregation of compute resources: *pooling of remote system memory*. ThymesisFlow implements a HW/SW co-designed memory disaggregation interconnect on top of the POWER9 architecture, by directly interfacing the memory bus via the OpenCAPI port. We use ThymesisFlow to evaluate how disaggregated memory impacts a set of cloud workloads, and we show that for many of them the performance degradation is negligible. For those cases that are severely impacted, we offer insights on the underlying causes and viable cross-stack mitigation paths.**

***Index Terms*—Disaggregation; OpenCAPI; POWER9;**

I. Introduction

In the constant race between delivering performance to customers and maintaining high energy efficiency, cloud systems have gone through many transformations: starting from an infrastructure composed of a series of computers connected to the network and practically rented to customers, down to an abstract infrastructure where the mapping between hardware and applications is hidden by an intermediate software layer. The latter, usually referred to as *Software-Defined Infrastructure*, is usually implemented via server virtualization, containerized applications and software-defined networking.

Ideally, software-defined resource orchestration aims at delivering, at runtime, exactly the amount of resources required by a given workload. In other words, the ultimate goal is to synthesize virtual hardware platforms with the right resource mix, without any over- or under-subscriptions. However, the current software-defined techniques (i.e., virtualization and containers), do not take into consideration computing resources, such as memory, CPUs, and accelerators, outside the physical server boundaries. The overall resource proportionality, that is fixed within each server, has to be carefully decided at installation time. For this reason, today it is common to have specialized machines tailored to specific types of workloads (CPU intensive, accelerator intensive, etc.) leading often to a fragmented system that is difficult to fully exploit. Given the aforementioned constraints, and taking into account that memory/CPU demand ratios, of typical cloud workloads, span across three orders of magnitude [1], [2], it becomes a complex scheduling problem to keep a high utilization of computing resources. To this end, the currently available software techniques, and scheduling approaches are not capable of delivering such a level of efficiency, targeted by the cloud providers, in the long run. Apart from the energy concerns, that heavily depend on the ability to improve utilization of a given infrastructure, the decoupling of hardware refresh cycles of the various hardware components, that today have to be updated concurrently, is most desirable.

Hardware-level disaggregation enables the novel organization of the data-centre infrastructure as a pool of heterogeneous resources (i.e. CPUs, memory, accelerators, network interfaces, and storage devices) that can be interconnected on-demand to form logical servers that closely match the requirements of a given workload. This paradigm provides several important benefits. First, it enables the components of a computing system to be practically integrated on the same bus, regardless of their physical location in the data-centre, thus significantly improving resource utilization (e.g., unused local system memory can be provided for use to a neighbour server). Second, in some cases, it eliminates the need for software-level, scale-out techniques which exchange data over the (sometimes over-subscribed) host network stack and, therefore, improves the actual data flow of a given workload. Last but not least, it unlocks the refresh cycles of hardware components in the data-centre.

Undoubtedly, the holy-grail of resource disaggregation is the *pooling of main system memory*, due to the stringent latency and bandwidth requirements of accesses to the main memory of the system. Additionally, from the System-on-Chip (SoC) main bus standpoint, every peripheral is memory-mapped (even if through a PCIe root complex bridge), and communicates with the rest of the integrated components with specific load and store transactions of the bus-architecture. Therefore, the disaggregation of the main system memory paves the

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00075

way for the hardware-level disaggregation of tightly integrated accelerators (like high-end GPUs) and co-processors.

Prior lines of research and recent commercial efforts introduce the memory disaggregation paradigm at the level of the operating system [3], [4], [5], [6], [7], [8]. The common mechanism that underpins all these approaches is the over-subscription of remote memory resources, that is combined with an OS trap, namely a page-fault. Even though the proposed mechanisms can take advantage of the fastest interconnect available, such as RDMA over Infiniband, there are still major concerns such as memory thrashing and host network stack over-subscription.

In this paper we introduce ***ThymesisFlow***, a software-defined - HW/SW co-designed datapath for materializing disaggregated memory. The design of ThymesisFlow addresses the key challenges of hardware-level disaggregation, and incorporates state-of-the art components into a full stack prototype enabling the evaluation of real workloads running on top of the IBM POWER9 [9] AC922 (the base platform of Summit supercomputer [10]). Taking into account emerging specifications like Gen-Z [11] that aim to provide a disaggregated I/O fabric, and building on top of the latest cache-coherent attachment technology for off-chip peripherals like OpenCAPI [12], NVLink2 [13] and the developing CXL [14] standard, ThymesisFlow design provides a high-performing all-hardware disaggregated memory solution.

There are many open research questions related to a fully hardware disaggregated system [15] for the cloud that are out of the scope of this paper. In this work, we focus on the feasibility of a memory disaggregated system, its performance evaluation, and valuable projections regarding scaling the size of our prototype and network infrastructure towards a production-grade disaggregated system. The main contributions of this paper are the following:

- Full hardware memory disaggregation prototype, with a software-defined out-of-band control plane and 100Gbit/sec network facing signaling.
- Run-time attachment/detachment of byte-addressable disaggregated memory to a running Linux Kernel exploiting dynamically created NUMA nodes to host the remote memory.
- End-to-end performance evaluation of the effects of memory disaggregation on a mix of real-world applications representative of typical cloud workloads.
- Discussion of the implications at the network level and scale of a real deployment of memory disaggregated system.

To the best of our knowledge, this is the first time a disaggregated architecture has been prototyped on commercially available hardware, and the first experimental evaluation of hardware-level memory disaggregation for real-world workloads on a complete software and hardware stack.

II. Motivation

One of the crucial factors, that cloud providers target to minimize, is total cost of ownership (TCO). Unlocking physical resource proportionality by disaggregating the compute and memory resources, results in workload defragmentation, improved resource utilization and, consequently, significant TCO reduction of a data-centre infrastructure.

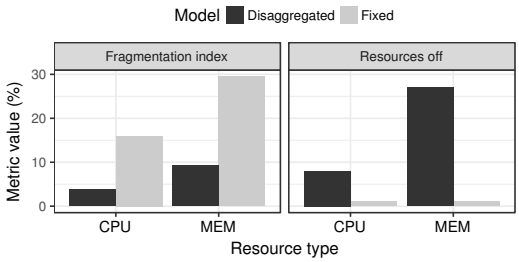


Fig. 1. Data-centre utilization using the Google Cluster Data Traces [16] for the conventional vs. disaggregated data-centre models. [left] Average resource fragmentation index (lower is better) [right] Average amount of resources that can be switched off (higher is better).

To quantify the benefits of computational resources disaggregation, we developed a custom tool that consumes entries from the publicly available Google ClusterData [16] trace and simulates resource allocation/deallocation requests for two data-centre infrastructures, namely a disaggregated and a traditional ("fixed") one. The "fixed" data-centre model comprises of 12555 servers, matching the configuration of the Google trace. Similarly, the disaggregated data-centre is modeled as a set of compute and memory modules such that in total it offers the same amount of resources as in the "fixed" case; specifically, we model 12555 compute and 12555 memory modules, with the total available memory spread evenly among the latter. Also, the disaggregated model assumes that each module connects to the data-centre interconnect via 16 links (modeling parallel transceivers), and that a fully connected topology enables any permutation of point-to-point connections between compute and memory modules. Both models employ a resource scheduler that uses an online best-fit allocation policy without resource overcommitment.

Using the aforementioned setup, we issued the workload allocation requests of the ClusterData trace for both data-centre models and we measured their respective level of resource fragmentation. This metric corresponds to the number of resources that must be kept powered on, even if they are not utilized by the scheduled workload (partially allocated hardware units). As shown in Fig. 1, for the "fixed" model, the fragmentation index is at 16% for CPUs and 29.5% for memory. However, for the disaggregated model there is a significant reduction compared to the "fixed" model (3.86% and 9.2% for CPUs and memory, respectively). As for potential energy saving, an average of 1% of servers is completely unused, and thus could be switched off, for the "fixed" model. The disaggregated model allows the scheduler to shut down 8% and 27% of the total number of the compute and memory modules, respectively. These quantitative findings are testimony to the promise brought by disaggregation as utilization and operational expense savings booster, largely forming the motivation of the present work.

III. Related Art

The idea of improving data-centre utilization via resource pooling and composable/disaggregated infrastructures

has been extensively explored in the literature. Some previous work have addressed general concerns around disaggregation such as: what is the scale that disaggregation makes sense; what is the best network technology to sustain a fully disaggregated system [15]. Some more recent work has explored operating systems aspects [17], and the possibility of a composable system where accelerators are remotely attached to machines via custom PCI switches [18], demonstrating that the same workloads mix could be served with a lower number of physical accelerators that are dynamically assigned to applications. Finally, the authors in [19] evaluate disaggregated memory with a SparkSQL workload, showing that — throughput-wise — memory disaggregation can be feasible even with conventional 40Gbps interconnects.

Breaking the monolithic design of data-centres 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 [20]. Notable demonstrators and prototype concepts include the Intel Rack Scale Design [21], 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 [22]. Similarly, the HPE "The Machine" [23] prototype showcases SoCs accessing remote memory via specialized bridging controllers and fabric (e.g., Gen-Z [11]). Our work shares common objectives with and can act complementarily to such and related designs; the 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, whose feasibility is proven by a hardware prototype.

Shared memory clusters have similar challenges and some of the technical and business objectives of disaggregated data-centres. Distributed shared-memory machines like NumaScale [24] or SGI UV [25] target parallel and distributed applications that require deployment on a large number of tightly cooperating cores; they typically build on distributed cache coherency protocols to turn underlying island domains into a single cache-coherent shared-memory machine. This approach trades flexibility at the cost of scalability because maintaining coherency becomes harder as the number of domains increases [26]. In contrast, our work targets typical cloud data-centre workloads where applications are developed to leverage socket-level (e.g., shared-nothing) parallelism and to make heavy use of in-memory computing, without tangible benefits from a shared memory scheme.

Looking at the core technology enabling memory disaggregation, previous approaches in the literature can be clustered in two main categories [27]: i) software-based, referred to in this paper as "*remote memory*" and ii) hardware based, herein referred to as "*memory disaggregation*".

Previous work on software *remote memory* often involves heavy modifications to core components of the OS (e.g., memory subsystem), and forces application to use custom libraries to gain access to the extended memory (e.g., page swap devices, RDMA transfers). Hotpot [5], Infiniswap [6] and the work by Lim et al. [28] all pursue the goal of memory disaggregation via a page-fault / swapping based approach. Compared to the above, our solution provides applications with transparent access to disaggregated memory in the form of raw byte-addressable memory (`ld/st` semantics). Existing applications and operating systems can directly benefit from our solution, without the need for modifications, custom middleware or intrusive OS extensions (e.g., swap and page fault -based data fetching). Furthermore, our approach uses a dedicated network to avoid impacting the main network latency with the traffic generated by memory disaggregation [29].

In the scope of hardware *memory disaggregation*, most of the approaches propose a combination of OS and micro-architectural extensions [30], devise extended addressing models embedding disaggregation information directly in the physical address [31], and often leverage dedicated programming models [32]. Our work does in part build upon ideas from the above efforts, e.g., by offering access to disaggregated memory

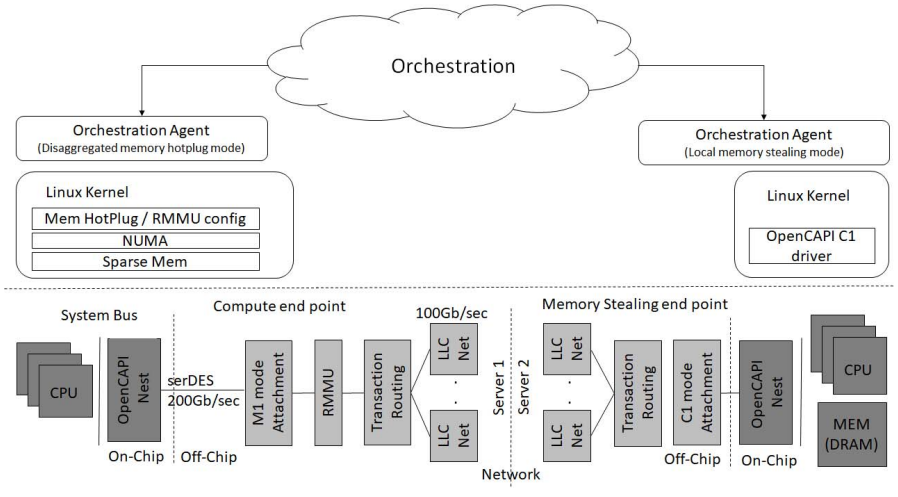


Fig. 2. ThymesisFlow overall architecture.

at load/store instruction granularity, similar to [30], [31]. Still, there are three key factors that strongly differentiate the present work: a) our approach does not require intrusive changes to the OS or dedicated programming models and is fully transparent to applications; b) this work presents results obtained on a real disaggregated memory prototype built on off-the-shelf hardware; c) we offer a complementary fresh look on full-fledged applications that are trending in cloud and hyperscale systems, while previous work is mostly focusing on multi-core application kernels [31].

IV. Architecture

Delivering an efficient hardware disaggregated memory system introduces several architectural challenges that need to be addressed, such as: i) appropriate interception of memory transactions, address translation and forwarding to the various remote destinations, ii) accurate design of the memory transaction routing scheme, the network interface and the topology for remote memory access; iii) seamless OS-level support for runtime attachment and detachment of disaggregated memory sections, and iv) support for software-based management of the disaggregated memory pool, including global state bookkeeping and allocation policies.

A. Hardware Interconnect Architecture

The ThymesisFlow hardware interconnect architecture (Fig. 2) features two different endpoint roles: i) *compute role* – this is the endpoint that introduces remote memory to a physical address space range on a host system (the recipient); ii) *memory-stealing role* – reserves a portion of the local memory on the host system (the memory donor) which is exposed, as disaggregated memory, to a neighbour host.

ThymesisFlow embraces the OpenCAPI [12] cache-coherent attachment technology, that is available today on IBM POWER9 processors, to both intercept memory transaction traffic and materialize the endpoint functionality.

The OpenCAPI specification allows off-chip peripherals to behave either as memory controller receiving cacheline traffic generated from the SoC processors (OpenCAPI M1 mode) or, as an accelerator that can directly master cache-coherent transactions to the virtual address space of an associated application. In the latter case, transactions are completed without the intervention of host processors or any DMA engine (OpenCAPI C1 mode). The ThymesisFlow compute and memory-stealing roles leverage the OpenCAPI M1 and C1 modes, respectively.

1) Compute endpoint: The disaggregated memory exposed via the ThymesisFlow compute endpoint is perceived as regular system memory. More specifically, the POWER9 firmware assigns at runtime a portion of the host real address space (i.e., physical address space) to the compute endpoint. Each transaction address is subject to a series of transformations before it can be forwarded to the memory-stealing endpoint, as depicted in Fig. 3. An effective address emitted at the compute side is first translated into a real address by the processor MMU. The real address is received by the ThymesisFlow device in its internal representation (the Device Internal Address Space is always starting from address 0x0). The internal address is finally translated into a valid effective address that can be emitted into the memory-stealing endpoint system bus. This translation task is undertaken by the ThymesisFlow Remote Memory Management Unit (RMMU) which is integrated into the compute endpoint.

The RMMU is designed to be used in conjunction with Linux kernel sparse memory model [33]. In this approach, the Linux kernel divides the physical address space assigned to the main system memory, into fixed-size aligned sections. Each memory section is independently handled by the kernel, and can be "hotplugged" at runtime to expand the available system memory. Similarly, the ThymesisFlow RMMU features a section table with one entry per section that contains: a) the address offset that must be applied to convert the transaction address from the internal device representation to the effective address of the memory-stealing counterpart and b) a network identifier that is added in the transaction header and is used by the routing layer. A specific bit range of the transaction address, common to all transactions belonging to the same section, serves as the table index.

The one-to-one mapping between Linux kernel sparse memory model and the ThymesisFlow RMMU configuration, defines the section as the minimum unit of disaggregated memory that can be independently handled. Each section is required to be associated to a consecutive effective address space of the same size at the memory-stealing side, guaranteeing that transactions belonging to the same section receive the same network forwarding information. The architecture logically groups all transactions (and their responses) in-transit between

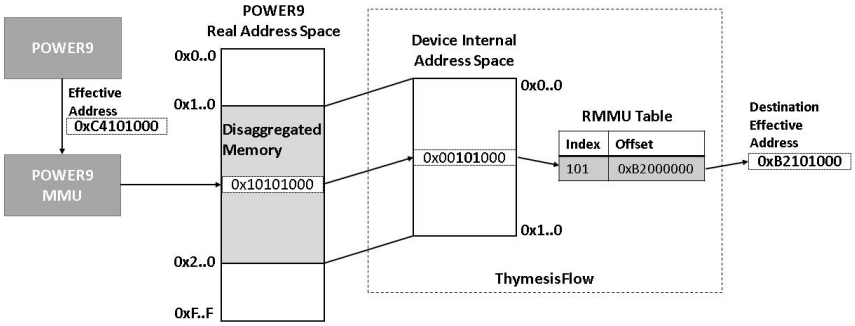


Fig. 3. ThymesisFlow address translation process.

a given compute and memory-stealing endpoint, and belonging to a specific section, as an *active thymesisflow*. Each active thymesisflow is associated with a unique network identifier. Finally, when a memory access transaction exits the RMMU of the compute endpoint, it has undergone all the modifications that are required to be mastered at the remote memory-stealing endpoint.

2) Memory-stealing endpoint: The memory-stealing endpoint enables access to the machine effective memory space from a remote compute node. A memory-stealing process allocates and pins to system memory the amount of cacheline-aligned memory requested by the remote node. The stealing process allows ThymesisFlow to access the memory reserved by registering its Process Addres Space ID (PASID) [34] with the memory-stealing endpoint hardware. The pointer (effective address) to the memory reserved is passed to the orchestration layer that calculates the proper offsets to be applied by the compute endpoint RMMU. At this point, the memory-stealing endpoint is passive and does not require further configuration: it does not modify the transactions, and does not need to receive any network information. The memory-stealing endpoint responds to transactions using the channel they arrived from, and the network identifiers that were already embedded in the arriving transaction headers.

3) Routing Layer: Right after the endpoint attachment module (Fig. 2), the ThymesisFlow stack features a routing layer to forward transactions towards remote endpoints. Each transaction is handled independently, based on the network information that is included in the header (added by the RMMU), and therefore the ThymesisFlow architecture allows any number of endpoints to be concurrently connected. In addition, the ThymesisFlow routing layer implements channel bonding: transactions belonging to an *active thymesisflow* can be forwarded using two or more physical network channels in a round-robin fashion. The bonding mode is enabled in-band by appropriate transaction header network identifiers on a per *active thymesisflow* basis. A network channel may be shared concurrently between different *active thymesisflows* regardless if one or more of them are using the channel in bonding mode. This approach enables the investigation of more sophisticated channel sharing approaches that go beyond simple round-robin, and will be able to offer bandwidth allocation and QoS capabilities.

4) Network facing stack: ThymesisFlow provides a reliable network facing channel by introducing a Link-Layer Control (LLC) protocol that implements the following features: i) backpressure support using a credit-based mechanism to protect Rx side from overflowing, and ii) frame replay support introducing a reliability scheme where a sequence of previously transmitted frames can be replayed in-order by the Tx side, upon request of the Rx side. ThymesisFlow LLC endpoints exchange credits by piggy-backing them on the transaction headers of requests and responses. Each credit represents an empty slot at the Rx ingress queue. The depth of the Rx ingress queues has been carefully calculated to avoid credits starvation at the Tx side. ThymesisFlow LLC frame replay is based on a custom framing scheme. All transactions from *active thymesisflows* that reach the LLC layer of a network channel are grouped in frames composed of a pre-defined number of flits. Incomplete frames are padded with single-flit `nop` transaction headers for immediate transmission, to avoid waiting for additional transaction flits to arrive. In addition, special single-flit frames are used as in-band messages to transfer replay requests to the Tx side. During link bring-up, the ThymesisFlow LLC Tx side agrees on a starting frame identifier with the Rx side and begins the in-order transmission of frames, incrementing the frame identifier accordingly after the transmission of each frame. If a frame is not received, or is received in error, the design triggers a frame sequence replay by exchanging appropriate in-band messages between the involved endpoints.

The ThymesisFlow LLC design features a 32B wide datapath which drives four datalink-layer bonded network facing transceivers. Notably, the LLC design does not impose any dependencies on the network MAC. Consequently, both a packet network (e.g. 100G Ethernet or EDR Infiniband) or circuit-based bit-for-bit network MAC can be used.

B. Operating System Support

The ThymesisFlow configuration space is exposed to the Linux operating system as a memory mapped I/O (MMIO) area, using the OpenCAPI generic device driver (Fig. 2).

A user-space *agent* runs as a daemon on every host, to issue the appropriate configuration commands received from the orchestration layer. The role of the user-space agent is twofold: i) configure the compute endpoint by performing the necessary operations required for physical and logical attachment of disaggregated memory or, ii) allocate local host memory and make it available to the memory-stealing endpoint. The logical attachment of disaggregated memory to a running Linux kernel is performed using the Linux *memory hotplug* [35] functionality, which was originally designed to plug and unplug local physical memory modules to and from conventional servers. The only information needed to hotplug a memory section is its start address in the physical address space where the compute endpoint is mapped. The orchestration software, aware of the global allocation mappings, passes this information to the agent, which uses the *memory hotplug* subsystem to probe and online the new memory.

At hotplug time, each disaggregated memory section is mapped to a CPU-less NUMA node [36], reflecting the respective transaction RTT delay between compute and memory-stealing endpoints. Thanks to this support, the kernel can optimize the access to frequently used memory areas by reusing existing NUMA page migration algorithms that move pages from distant to closer (including local) memory nodes [37].

C. Control plane

The control plane implements the software-defined mechanisms that allow dynamic attachment of disaggregated memory to the compute nodes in the system (Fig. 2). The main responsibilities of the ThymesisFlow control plane are: i) system state maintenance, ii) configuration of ThymesisFlow endpoints and possible intermediate switching layers, iii) system access interface, and iv) security and access control.

The system state is modeled as an undirected graph whose nodes are compute and memory endpoints, transceivers associated with each endpoint and switch ports. The edges of

the graph are instead the possible physical links between nodes. For each disaggregated memory allocation request, the control plane traverses the graph looking for the best available path connecting the compute and memory stealing endpoints involved. Once a suitable path is found and its resources are reserved, the control plane generates the suitable configurations and pushes them to the appropriate agents. We use Janusgraph [38], a distributed graph database, as the backend to the ThymesisFlow control plane.

The various remote memory allocation/deallocation interactions occur via a REST API. An access control system ensures that only users with enough privileges can act on the system status. This interface can be used directly by administrators to build ad-hoc computing platforms. As part of the future work we plan to integrate this interface with existing cloud orchestration frameworks such as OpenStack [39] or Kubernetes [40], to implement transparent resource allocation based on incoming VM or containers instantiation requests.

Finally, the control plane guarantees safe distribution of remote memory resources. This is enforced by using compute endpoint configurations allowing memory transactions forwarding only towards legal destinations, and fail otherwise. To make sure no malicious software can push illegal configurations, trusted node agents and network elements firmware accept configuration updates only from a trusted control plane.

V. System prototype

ThymesisFlow is a fully functional prototype and implementation of the hardware, OS and orchestration software architecture presented in Section IV, and has been developed on top of the OpenCAPI [12] FPGA stack available for IBM POWER9[9]. ThymesisFlow hardware datapath is tightly designed in Verilog. The experimental prototype is composed of three IBM Power System AC922 nodes [41]. Each node features a dual socket POWER9 CPU (32 physical cores and 128 parallel hardware threads) and 512GB of ram. Two of the nodes are equipped with an Alpha Data 9V3 card [42] that features a Xilinx Ultrascale FPGA. The ThymesisFlow datapath leverages one OpenCAPI FPGA stack instance that interfaces a POWER9 processor at $200Gbit/sec$ by bonding $8x$ GTY transceivers at $25Gbit/sec$. For the network, the prototype is using Xilinx Aurora protocol [43] that provides a low-latency and basic framing datalink layer with CRC support. The aurora-based network-facing pipelines are organized in two totally independent channels, each one driving $4x$ bonded GTY transceivers at $25Gbit/sec$ ($100Gbit/sec$). Therefore, the ThymesisFlow design features three mesochronous clock domains (one for each transceiver group) that all run at $401Mhz$.

The network facing QSFP28 cages are connected with direct attached cables to provide point-to-point and point-to-multipoint configurations. The hardware datapath flit RTT latency of this prototype is roughly $950ns$ which includes four crossings of the FPGA stack and six serDES crossings (2x at compute endpoint side, two for the network and two at the memory stealing endpoint side).

On the software side, all systems run Linux kernel version 5.0.0 with memory hotplug [44] and NUMA extensions, as described in Section IV-B. Last but not least, the ThymesisFlow hardware design and relevant software are all available under the OpenPower [45] initiative to the opensource community [46] to accelerate research on hardware-level disaggregation.

VI. Evaluation

We evaluate the ThymesisFlow prototype and analyze the impact of disaggregated memory by measuring the end-to-end performance of various cloud applications. In the rest of this section, we present the experimental setup, discuss our evaluation methods, and present our results.

A. Experimental setup

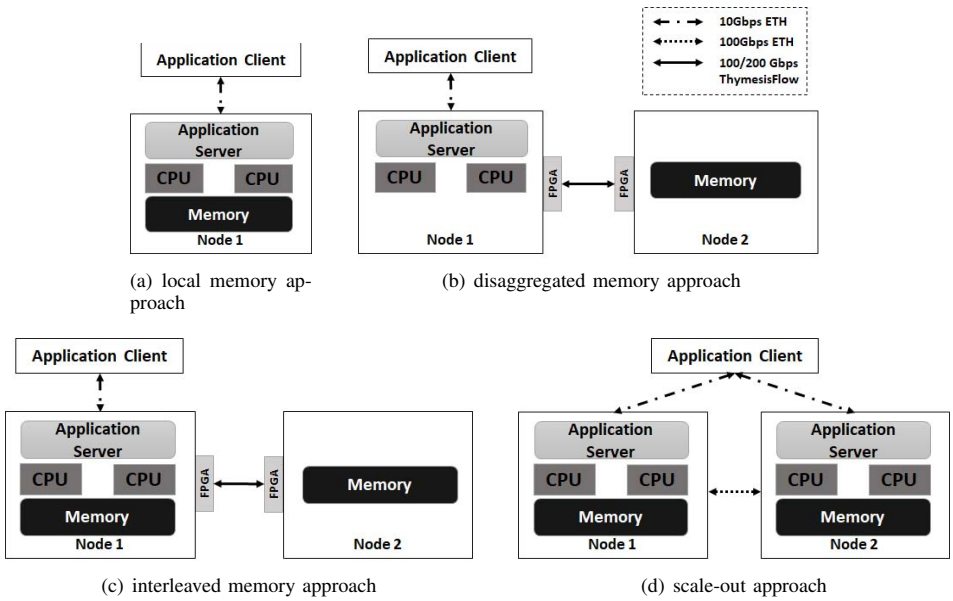
Our evaluation was performed using the three servers described in Section V. The two nodes featuring an FPGA, provide the memory disaggregation capabilities and run the server-side of the applications under evaluation. The third node executes the respective clients of the applications.

In our evaluation we have investigated four different system configurations, as shown in Fig. 4: i) *local*: all memory requests are served locally on the same node where the application server is running (Fig. 4(a)); ii) *single-disaggregated*: all memory needs of the application server are satisfied by memory stolen from the neighbour node, and only one ThymesisFlow network channel (100Gb/s) is used (Fig. 4(b)); iii) *bonding-disaggregated*: similar to *single-disaggregated* but both ThymesisFlow network channels (200Gb/s) are used (Fig. 4(b)); iv) *interleaved*: memory pages are allocated on both local and disaggregated memory following a round-robin policy (Fig. 4(c)); v) *scale-out*: this is the traditional, widely-adopted, approach to scale cloud applications. In contrast to the previous configurations, the application server is scaled on the two available nodes and memory allocations are always local to the nodes (Fig. 4(d)). Note that, while the total amount of memory remains unchanged in all the configurations evaluated, when comparing disaggregation vs scale-out, the disaggregated configuration has half the number of CPU cores available compared to the scale-out.

Moreover, for the *scale-out* configuration, the two "server" nodes are connected with each other via 100Gb/s Ethernet. For the remaining configurations the "server" nodes are connected through ThymesisFlow (copper cables) at 100Gb/s, or 200Gb/s in the case of *bonding-disaggregated*. Also, the "client" machine is connected to the "server" machines via 10Gb/s Ethernet, for all configurations.

B. Application selection

For each of the previously described system configurations, we have performed a double set of experiments targeting both the raw memory bandwidth achievable with our prototype, as well as the evaluation of three application classes that we deem to be representative of the large variety of cloud workloads executed today. All applications selected are free and open-source, and occupy a large-enough area on the resource proportionality continuum (i.e., exhibiting a large heterogeneity in terms of memory/CPU/IO usage ratios).



(a) local memory approach

(b) disaggregated memory approach

(c) interleaved memory approach

(d) scale-out approach

Fig. 4. Resources assigned to the application servers on the compute nodes for each experimental configuration.

- *Sustainable memory bandwidth*. We evaluate ThymesisFlow bare performance using the STREAM benchmark [47], the de facto industry standard to measure sustainable memory bandwidth and overall processing balance, as perceived by user space applications.
- *In-memory database*. They are widely adopted, for example, in on-line commercial systems. We use the NewSQL VoltDB in-memory database [48] (v6.5.8 - community edition) to evaluate this class of cloud applications.
- *In-memory application-level caching*. Application caches are widely deployed in data-centres to speed up key-based data retrieval, e.g., in Web applications as a layer in front of relational databases. We evaluate Memcached [49] (v1.5.20) as a frequently used representative of this application class.
- *Data analytics*. With this term we refer to the large class of applications that execute complex queries over large data bases [50]. A common characteristic of these workloads is a marked presence of I/O activity (mostly cpu-based). We chose the Elasticsearch [51] (v7.4.1) to study this category.

C. Sustainable memory bandwidth

We configured STREAM to use 160 million array elements, requiring a total memory of 3.66 GiB, which is well beyond the system cache size. Each benchmark run executes four kernels, i.e., "copy", "scale", "add" and "triad" [47]. 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; "add" 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 4, 8, and 16 hardware threads. Leveraging the local and remote NUMA domains, we repeat the same executions using all the configurations depicted in Fig. 4.

Fig. 5 shows the results of our evaluation, in terms of sustained memory bandwidth (GiB/s), for each system configuration as clustered bars. For the "copy" kernel, we observe that with 4 threads the *single-disaggregated* configuration can achieve $\sim$ 10 GiB/s bandwidth towards disaggregated memory, reaching close to the theoretical maximum of 12.5GiB/s when using 8 threads. As we increase concurrency, in terms of threads number, performance decreases because the network facing stack gets closer to the saturation threshold, as the ThymesisFlow network is already operating at its peak (100Gib/s). That said, instinctively, someone would expect the *bonding-disaggregated* configuration to deliver approximately $2\times$ the performance of the *single-disaggregated*. However, this is not the case, because the OpenCAPI mode C1 used for the memory side of our prototype works with 128B transactions, and can exploit a relatively small burst size. This leads to a

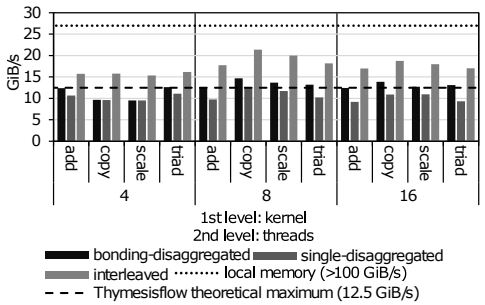


Fig. 5. STREAM benchmark performance comparison.

maximum actual bandwidth to/from memory in the range of 16GiB/s, which is in line with the $\sim$ 15GiB/s obtained with *bonding-disaggregated* for the "copy" benchmarks running with 8 threads. Notably, the OpenCAPI C1 mode has been measured to achieve 20GiB/s by leveraging 256B memory transactions, which cannot be used in the current ThymesisFlow design as the POWER9 processor is only issuing 128B wide ld/st transactions (cache line size). After an initial analysis, we have assessed that merging 128B transactions (at both the compute or memory endpoint) into 256B ones would greatly increase the complexity of our design. Consequently, the benefits from using bigger data bursts would be minimized. Overall we measure a $\sim 30\%$ improvement for the *bonding-disaggregation* configuration.

Notably, the interleaved configuration is outperforming all the other configurations. In this configuration, the Linux kernel is alternating on a 50/50 basis pages from the two NUMA nodes. This results in 50% of local and 50% disaggregated allocations of memory pages. Such synergy of local and disaggregated memory mitigates the higher access latency of disaggregated memory and provides a solid hint on what to expect from a real application, where the mix of CPU and memory instructions is more complex and cache locality is better exploited. These effects are not visible with STREAM because it is a memory intensive benchmark, where at each iteration of the kernels there are always more memory than CPU instructions.

Even though these results highlight the huge difference in bandwidth between local and disaggregated memory, they also show how a synergistic approach can dramatically maximize memory bandwidth. This experiment also confirms that ThymesisFlow pipelined design and OpenCAPI FPGA stack are capable of exploiting almost the whole bandwidth available between the nodes, as well as between FPGA and CPU.

In addition, STREAM is designed for measuring sustained memory bandwidth and does not contain the mix of CPU, I/O and memory instructions that can be observed in a real application. In the next sections we show how the performance of our ThymesisFlow prototype is appealing for real cloud applications where internal and network-based synchronization barriers exist and take their toll at the delivered performance.

D. In-memory database

VoltDB[48] is an in-memory and fully ACID-compliant (Atomicity, Consistency, Isolation, Durability) RDBMS. More specifically, VoltDB is based on H-Store [52] and is designed as a *share-nothing* architecture. Such a design pattern, allows VoltDB to offer built-in support for horizontal scaling and multi-node deployments. VoltDB splits data, SQL tables in this case, into *partitions* and assigns their corresponding processing to individual threads. Consequently, the data partitions can be distributed across different CPU cores or even exceed the physical boundaries of a node and form a multi-node cluster.

For this evaluation, we used the Yahoo! Cloud Serving Benchmark (YCSB) [53] suite to extract the performance of VoltDB for all the experimental configurations presented in Section VI-A. YCSB is a workload generator client targeting benchmarking of data serving systems via six different workloads, namely A. B, C, D, E and F. The YCSB workloads can be divided into two groups: i) *Read intensive*: workloads with $> 95\%$ read transactions. This is the case of workloads B, C, D and E ii) *Mixed*: all the transactions generated by each client are split in 50% reads and 50% other transactions (e.g., updates). This is the case for workloads A and F. The reader is referred to the YCSB documentation for a detailed description of all the available workloads [54].

To better understand the behavior of VoltDB for different number of partitions — more partitions translate to an increase in parallelism — we profiled VoltDB for all YCSB workloads with 2000 YCSB client threads. All the results are collected using Linux `perf` tools [55]. More specifically, Fig. 6 shows the average retired Instructions per Cycle (IPC) across the whole CPU package and the average number of utilized CPU cores (UCC) for the VoltDB process. This experiment was repeated for all YCSB workloads, with the *local* and the *single-disaggregated* setups.

The average UCC is based on the *task-clock* perf event that reports how parallel each task is, by counting how many CPU

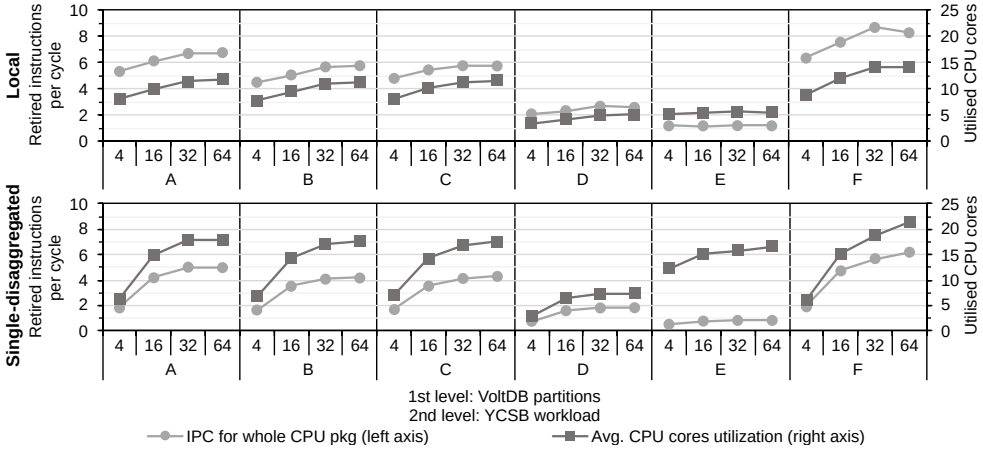


Fig. 6. VoltDB profiling for all YCSB workloads under various number of data partitions. The lines with circle markers show the IPC, while the lines with square markers show the UCC.

cores were utilized during its execution. For the estimation of the average IPC across the whole CPU package, we used the *instructions* and *cycles* perf events. The *instructions* perf event reports the amount of retired instructions, while *cycles* reports the number of CPU cycles. The single-thread IPC of the VoltDB process is obtained by dividing *instructions* by the value of *cycles*. Finally, the average IPC across the whole CPU package is obtained multiplying the single-thread IPC by the average UCC. During our experiments, we also capture the *stalled-cycles-frontend* and *stalled-cycles-backend* perf events, to provide extra information about the reported IPC of our profiling campaign.

Interestingly, for the local configuration (Fig. 6) we observe that increasing the number of partitions and, thus, scaling VoltDB horizontally, results in an increased IPC for workloads that are not dominated by READ operations, such as workload A and F. However, the biggest improvement is observed when we increase the number of data partitions from 4 to 16. For higher partition numbers, the IPC gains remain relatively small. On the other hand, when READ or SCAN operations dominate the workload, such as workload B, C, D, and E, the horizontal scaling of VoltDB does not significantly increase the IPC. To further investigate this behavior we measured the network bandwidth utilization between the YCSB client and the VoltDB host system, and we found that the network was not saturated. Also, we lowered the number of YCSB client threads from 2000 to 500, and VoltDB exhibits the same behavior.

On the other hand, for the disaggregated memory setup (Fig. 6) , we observe higher IPC and higher UCC when increasing the number of VoltDB data partitions. In light of the data collected this effect is to be associated with the higher latency of disaggregated memory accesses that is relaxing the synchronization between threads across different data partitions. This results in a lower number of threads yielding the CPU while synchronizing and thus, into a higher average UCC. However, the increased latency in the *single-remote* has a negative impact on the IPC measured. In particular, for low partition numbers, such as 4, we observe that this latency heavily penalizes the IPC, because the synchronization overhead, mentioned above, is relatively smaller compared to the latency of the disaggregated memory.

However, for higher partition numbers (16 or more), we see that the IPC of the disaggregated setup increases but remains lower compared to the local memory configuration. This is also proven by the increased number of CPU back-end stalls measured in the *single-remote* configuration. A *back-end stall* happens when the pipe-line is waiting for resources (e.g., memory) or for a long latency instruction to complete. Overall we have measured that, for the *local* configuration, on average 55.5% of cycles resulted in a back-end stall. This number raises to 80.9% in the *single-remote* configuration. The above analysis shows the effect of memory disaggregation from an architectural standpoint, that for space reasons will not be presented for the other applications studied in this paper. However, the main goal of this work remains to evaluate the impact of disaggregated memory on applications performance. In other words, measuring how and if applications performance metrics (e.g., as throughput or transactions latency) are being affected from disaggregated memory accesses.

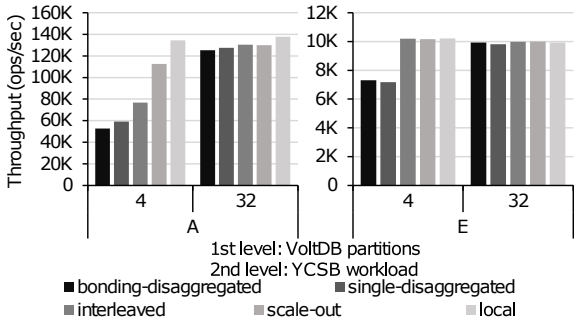


Fig. 7. YCSB workloads A and E throughput for all experimental setups (Section VI-A) under various number of data partitions.

Based on the similarity of YCSB workloads, in terms of IPC (Fig. 6) and due to space limitations, we present the application performance results only for workloads A and E for 4 and 32 VoltDB data partitions. More specifically, Fig. 7 shows the respective throughput, as reported by the YCSB client, for all the experimental setups presented in Section VI-A. Initially, we observe, that when running with 4 VoltDB data partitions all configurations using our ThymesisFlow prototype (i.e., *single-disaggregated*, *bonding-disaggregated*, *interleaved*) have a performance in terms of throughput that is significantly lower than the *local* and *scale-out* configurations. This is because of the aforementioned latency impact, introduced by our architecture, and the contention between the VoltDB data partitions.

In general, and as expected, the *local* configurations exhibits the best performance regardless of the workload and number of partitions. For workload A, all the remaining configurations such as *scale-out*, *interleaved*, *single-* and *bonding-disaggregated* are slower by 5.95%, 5.62%, 7.97%, 10.03%, respectively. The *scale-out* configuration is the second-fastest setup, as half of the VoltDB memory is allocated locally, on the host system, and the other half is stolen from a remote node. Interestingly, the performance of *scale-out*, *single-remote* and *bonding-remote* configurations remain similar, as the overhead of both local and network synchronization across data partitions for the *scale-out* setup is comparable to the latency of disaggregated memory accesses. Furthermore, for workload E, we observe that throughput is similar for all configurations, as the vast number of READ operations saturate the performance of VoltDB. However, the configurations that utilize our ThymesisFlow prototype have a similar performance with *local* and *scale-out* configurations. Notably, the *interleaved*, *single-remote* and *bonding-remote* use half computing resources (CPU) compared to the *scale-out* configuration. This is a solid indication that, for some applications, our architecture can replace the traditional scaled-out approach, while delivering a comparable performance with a reduced computing resources footprint.

E. In-memory application-level caching

We have chosen Memcached [49] as representative for this applications category since it is well known and widely used

in real cloud deployments. Memcached is an application-level in-memory cache exposing a simple TCP (or UDP) interface that allows to load (`GET`) or store (`SET`) key-value pairs. Typically, in-memory caches exhibit limited CPU utilization while performing many small read-only memory accesses.

To evaluate the performance of Memcached, we used the statistical models, of prior work [56], for the "ETC" traces and we developed a realistic load generator. To this end, these models are based on a comprehensive characterization of Memcached workloads in Facebook data-centres.

At startup time, the load generator "warms-up" Memcached by generating a number of `SET` requests that are large enough to fill the cache up to a configurable total size (10 GiB in these experiments). After the warm-up phase has ended, the load generator spawns 64 threads that act as Memcached clients posting `GET`/`SET` requests with a ratio of $30:1$ [56]. The key for each request is chosen from the key-space according to a Zipf distribution with configurable exponent, following the observations in [57]. The load generator records individual response latencies for each request and stops after each thread has issued 1 million requests. The key-value space size is 15 GiB and the Zipf exponent is set to 1.0. With this setup, we obtain an average hit ratio varying from 80% to 82%, close to the 81% value reported in [56].

Fig. 8 shows the cumulative distribution of measured `GET` requests latency for all the configurations described in Section VI-A. Results for `SET` requests follow the same trends, and are not shown due to space limitations.

From our evaluation, the *local* configuration presents the best performance with an average response latency of 600μs. This setup offers also the higher level of consistency with 90% of all requests served with only 19% degradation compared to the average latency. In average *interleaved*, *single-* and *bonding-disaggregated* configurations present similar response latency of respectively 614, 635 and 650μs. However, we register a higher level of latency degradation amounting to respectively 33%, 34% and 64% when serving 90% of all requests. For the *scale-out* configuration, we employ Twemproxy [58]; a proxy for the Memcached servers. Twemproxy supports the Memcached communication protocol and targets to reduce the number of open connections to the cache servers. Moreover, by employing a proxy, we simulate an environment, matching the one found in a typical data-centre, where the internal network of servers is not exposed to the various clients. Given the increased number of hops required to reach the service because of the proxy, *scale-out* registers an increased average latency of 713μs with up-to $2\times$ latency degradation when considering 90% of the requests.

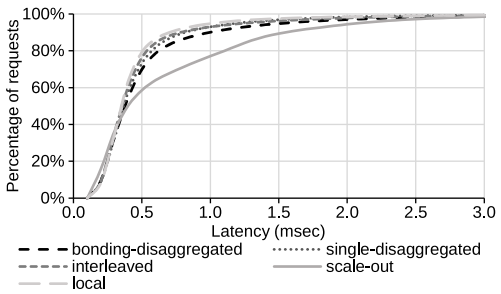


Fig. 8. Memcached GET transactions latency CDF.

Our results show that the configurations that utilize our ThymesisFlow prototype, offer similar performance to local configuration with an average increase in latency of up-to 7%. This is because Memcached exhibits a remarkably cache-friendly behavior [57], [56] due to the high spatial and temporal locality in its access patterns. The *scale-out* configuration demonstrates how a memory demanding workload can highly benefit from disaggregated memory. More specifically, for the *scale-out* configuration, the cost of network synchronization exceeds the cost of accessing a disaggregated memory and, consequently, the scale-out configuration results in an increase of transactions latency, 8% on average, and a much higher variability.

F. Data analytics

Elasticsearch [59] is an open-source search and analytics engine that supports multiple types of data such as textual, numerical and structured. Moreover, Elasticsearch is an extra layer, on top of Apache's Lucene [60], that supports indexing and automatic type guessing and utilizes a JSON-based REST API to expose certain features. Through this API, Elasticsearch can be instructed to perform several operations namely, search, update, insert, etc.

Furthermore, Elasticsearch saves all the data in the form of JSON-expressed documents. Documents with similar characteristics are part of an Index collection. Elasticsearch provides the ability to subdivide any Index into pieces, namely *shards*. Each shard is a fully-functional and independent Index that can be distributed across different cores of a CPU and even different physical servers (nodes), forming a multi-node cluster of Elasticsearch. However, on each node, despite the number of shards, all operations queue up into the corresponding thread pool based on their type. In particular, each Elasticsearch node holds several dedicated thread pools for different operations, that allow requests to be kept instead of being discarded.

In our evaluation we use ESRally which is an official Elasticsearch benchmarking suite, offering multiple benchmarking scenarios in the form of tracks. We have selected the "nested" track, consisting of various operations in the form of "challenges". The working dataset is a dump of StackOverflow posts retrieved as of June 10, 2016. Moreover, we experiment with various numbers of shards, such as 5, 32. Due to space limitations, in Fig. 9, we present only the results of the following challenges: i) RTQ: searches for all questions that feature a random generated tag; ii) RNQINBS: searches for questions (similar to RTQ) that contain at least 100 answers before a random date; iii) RSTQ: searches for questions (similar to RTQ) and sorts them in a descending fashion according to their date; iv) MA: queries all questions. Details of the remaining "nested" challenges can be found in the official ESRally documentation [61].

As shown in Fig. 9, the performance of each configuration strongly depends on the type of operations issued in each challenge. More specifically, for the RTQ challenge and *scale-out* configuration, Elasticsearch benefits from the extra computational resources, in the form of extra shards, and outper-

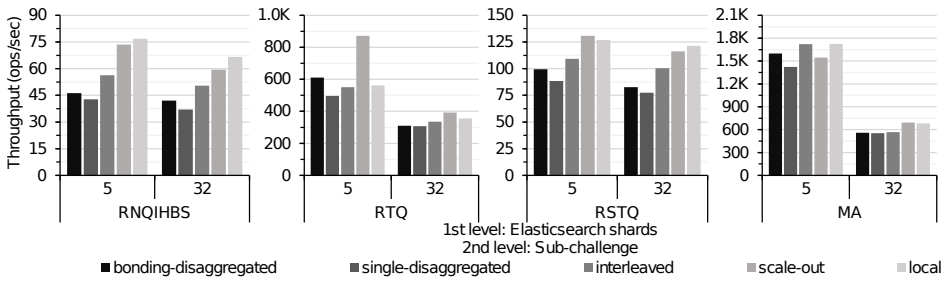


Fig. 9. "nested" ESRally track performance for all memory configurations.

forms any other configuration, including *local*. As expected, in this case, all setups that utilise the ThymesisFlow prototype, such as *interleaved*, *bonding-disaggregated* and *single-disaggregated* are slower by 58.33%, 42.65% and 75.65%, respectively. Similarly, for the challenges that require tighter synchronization across Elasticsearch shards (shards scaling results in a throughput degradation), such as RNQINBS, RSTQ, and MA, we observe that the *scale-out* configuration outperforms the *interleaved*, *bonding-disaggregated* and *single-disaggregated* configurations by 17.95%, 41.26%, 60.61% on average, respectively. However, for the MA challenge, the configurations that utilise our architecture offer similar performance with the *local* and *scale-out* ones. Our evaluation of Elasticsearch shows that there are cases for which the traditional scale-out approach clearly outperforms our design but, depending on the workload, memory disaggregation still offers appealing performance.

VII. Challenges And Future Work

While the ThymesisFlow FPGA-based prototype achieves acceptable RTT latency that is below $1us$, there is a considerable margin for improvement if the design was efficiently integrated in the processor SoC. First, at the architecture-level, the SoC transceivers could be driven by an appropriately modified design to directly interface the network, rather than the middle off-chip peripheral approach used in the current design, which would save four serDES crossings. Second, if ThymesisFlow was entirely implemented on an ASIC, several steps like the Physical Coding Sublayer (PCS) of the serDES stack would have considerably less impact on latency as well. Apart from reducing the hardware datapath latency, remote memory access experience can be further improved both with Operating System level approaches that are currently being investigated in the context of disaggregation, as well as by the introduction of an appropriate caching layer at the hardware-level (e.g. using HBM intermediate memory as cache). Sustainable bandwidth is also a consideration but it poses less challenges than latency and can be tackled (to a certain extend) by simply adding more hardware. For example, the IBM POWER9 processor has already four OpenCAPI stacks that collectively provide $800Gbit/sec$ of bandwidth and there are two such processors on the AC922 server.

Nevertheless, the network architecture poses the most important challenges, and with the currently available technologies, only rack-scale dissaggregation seems a feasible solution (i.e. at most one switching layer) to maintain the RTT latency to appropriate levels. At the scale of one or a few racks, a circuit switched optical network would be attractive. This would guarantee enormous bandwidth and absence of congestion. However, a circuit-switched network limits the scalability of the whole infrastructure as it remains limited by the number of ports available on each node, unless the switch is rapidly re-configured at the scale of packets or flows to serve another circuit on those ports. With a packet-based network on the other hand, a node could access all other nodes in the rack with no need for reconfiguration, although packet networks come with congestion issues as network links are shared between many connections. Many new technologies are being developed that could improve the performance of both solutions, such as all-optical switches at ns- or μs-scale [62], [63], [64], [65] and network interfaces at 200-400Gb/s or beyond [66], [67]. For both optical and packet-based networks, there is a tradeoff of application performance, resource utilization and total cost of ownership (CAPEX and OPEX). The authors recognize that scaling the rack and datacentre interconnect is one of the main technical and economic challenges in pursuing disaggregated systems and will proceed to explore possible avenues in this space. The reader is referred to previous work dedicated to this subject [68], [69], [70], [29].

VIII. Conclusions

In this paper we have presented ThymesisFlow, a hardware-level memory disaggregation prototype based on commercially available, state-of-the-art hardware components. The ThymesisFlow full stack approach attempts to strike a balance between performance and software-defined control, including the ability to synthesize and teardown logical servers at runtime - the main promise of disaggregated resource pooling. Evidently, hardware-level disaggregation comes with a performance tradeoff that needs OS-level support to be further minimized. However, we demonstrate that some important cloud workloads exhibit already an acceptable performance.

As transceiver technology evolves to support higher data rates and lower latency, the potential of hardware-level disaggregation will be stronger and beyond memory scale-up, to include the dynamic formation of CPU SMP domains and the direct communication of cache coherently-attached remote accelerators.

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] C. Reaño, J. Prades, and F. Silla, "Analyzing the Performance/Power Tradeoff of the rCUDA Middleware for Future Exascale Systems," vol. 132, pp. 344 – 362. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731519303491

[4] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont, "Welcome to Zombieland: Practical and Energy-Efficient Memory Disaggregation in a Datacenter," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. Association for Computing Machinery, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3190508.3190537

[5] Y. Shan, S.-Y. Tsai, and Y. Zhang, " Distributed Shared Persistent Memory ," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 323–337. [Online]. Available: http://doi.acm.org/10.1145/3127479.3128610

[6] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient Memory Disaggregation with Infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 649–667. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[7] yellow bricks.com, " VMWare Cluster Memory ," Online: http://www.yellow-bricks.com/2019/09/02/vmworld-reveals-vmware-cluster-memory-octo2746bu/, accessed: March 2020.

[8] W. Cao and L. Liu, "Hierarchical Orchestration of Disaggregated Memory," *IEEE Transactions on Computers*, 2020.

[9] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, " IBM Power9 Processor Architecture ," *IEEE Micro*, vol. 37, no. 2, pp. 40–51, Mar. 2017. [Online]. Available: https://doi.org/10.1109/MM.2017.40

[10] ORNL, "Summit Supercomputer," Online: https://www.olcf.ornl.gov/summit/, 2018, accessed: March 2020.

[11] G.-Z. Consortium, "Gen-Z Specification," Online: http://genzconsortium.org, 2017, accessed: January 2019.

[12] O. Consortium, "OpenCAPI Specification," Online: http://opencapi.org, 2017, accessed: January 2019.

[13] N. Corporation, "NvLink Interconnect," Online: http://www.nvidia.com/object/nvlink.html, 2018, accessed: January 2019.

[14] C. Consortium, " Compute Express Link ," Online: https://www.computeexpresslink.org/, accessed: February 2020.

[15] A. Roozbeh, J. Soares, G. Q. Maguire, F. Wuhib, C. Padala, M. Mahloo, D. Turull, V. Yadhav, and D. Kostić, "Software-Defined "Hardware" Infrastructures: A Survey on Enabling Technologies and Open Research Directions," *IEEE Communications Surveys Tutorials*, vol. 20, no. 3, pp. 2454–2485, thirdquarter 2018.

[16] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google Cluster-Usage Traces: Format + Schema," Google Inc., Mountain View, CA, USA, Technical Report, Nov. 2011, revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[17] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation," pp. 69–87, Oct. 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/shan

[18] I.-H. Chung, B. Abali, and P. Crumley, " Towards a Composable Computer System ," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPC Asia 2018. Association for Computing Machinery, pp. 137–147. [Online]. Available: https://doi.org/10.1145/3149457.3149466

[19] 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

[20] O. C. Project, " OCP Summit IV: Breaking Up the Monolith ," Online: http://www.opencompute.org/blog/ocp-summit-iv-breaking-up-the-monolith/, 2013, accessed: January 2019.

[21] I. Corp., " Intel Rack Scale Design ," Online: http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design/rsd-vision-brochure.html, 2017, accessed: January 2019.

[22] L. Inc., " Liqid Hyperkernel ," Online: https://liqid.com, 2017, accessed: January 2019.

[23] The Next Platform, " HPE Powers Up The Machine Architecture ," Online: https://www.nextplatform.com/2017/01/09/hpe-powers-machine-architecture/, 2017, accessed: January 2019.

[24] E. Rustad, " NumaConnect: A High Level Technical Overview of the NumaConnect Technology and Products (NumaScale Whitepaper) ," Online: https://www.numascale.com/numa_pdfs/numaconnect-white-paper.pdf, 2014, accessed: January 2019.

[25] SGI UV, " The World Most Powerful In-Memory Supercomputers ," Online: http://www.sgi.com/products/servers/uv/index.html, 2017, accessed: August 2017.

[26] M. A. Heinrich, " The Performance and Scalability of Distributed Shared-memory Cache Coherence Protocols ," Ph.D. dissertation, Stanford, CA, USA, 1999.

[27] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, " Remote Memory in the Age of Fast Networks ," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 121–127. [Online]. Available: http://doi.acm.org/10.1145/3127479.3131612

[28] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level Implications of Disaggregated Memory," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012, pp. 1–12.

[29] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, " Network Requirements for Resource Disaggregation ," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 249–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026897

[30] 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

[31] 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

[32] 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

[33] lwn.net, "Linux Kernel Sparse Memory Model," Online: https://lwn.net/Articles/789304/, 2019, accessed: March 2020.

[34] "PASID," Online: https://pcisig.com/specifications/pciexpress/specifications/ECN-PASID-ATS-2011-03-31.pdf?speclib=pasid, accessed: March 2020.

[35] kernel.org, " Linux Memory Hotplug Documentation ," Online: https://www.kernel.org/doc/Documentation/memory-hotplug.txt, 2017, accessed: January 2019.

[36] "Linux NUMA Documentation," Online: https://www.kernel.org/doc/Documentation/vm/numa, 2017, accessed: January 2019.

[37] R. Van Riel and V. hegu, "Automatic NUMA Balancing," Red Hat Summit 2014., 2014, accessed: January 2019.

[38] JanusGraph Authors, "JanusGraph," Online: https://docs.janusgraph.org/, 2020, accessed: April 2020.

[39] R. C. Computing, " OpenStack ," Online: https://www.openstack.org, 2018, accessed: January 2019.

[40] T. L. Foundation, " Kubernetes: Production-Grade Container Orchestration ," Online: https://kubernetes.io, 2018, accessed: January 2019.

[41] IBM, "IBM Power System AC922," Online: https://www.ibm.com/uk-en/marketplace/power-systems-ac922, accessed: February 2020.

[42] A. Data, "Alpha Data 9v3," Online: https://www.alpha-data.com/dcp/products.php?product=adm-pcie-9v3, 2017, accessed: January 2019.

[43] Xilinx, "Aurora 64B/66B v10.0 - LogiCORE IP Product Guide," 2015.

[44] A. Reale and M. Bielski, "Memory Hotplug Support for Arm64 — Complete Patchset v2," Online: https://lkml.org/lkml/2017/11/23/182, 2017, accessed: January 2019.

[45] O. Foundation, " OpenPower ," Online: https://www.openpower.org, accessed: March 2020.

[46] ThymesisFlow Developers, "ThymesisFlow," Online: https://github.com/OpenCAPI/ThymesisFlow, accessed: March 2020.

[47] 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.

[48] M. Stonebraker and A. Weisberg, "The VoltDB Main Memory DBMS," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, 2013.

[49] Dormando, "Memcached - a Distributed Memory Object Caching System," Online: https://memcached.org, 2017, accessed: January 2019.

[50] S. Chaudhuri and U. Dayal, " An Overview of Data Warehousing and OLAP Technology ," *SIGMOD Rec.*, vol. 26, no. 1, pp. 65–74, mar 1997.

[51] Elasticsearch, a distributed search and analytics engine, "Elasticsearch," Online: https://www.elastic.co, 2020, accessed: March 2020.

[52] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System," *Proc. VLDB Endow.*, vol. 1, no. 2, p. 1496–1499, Aug. 2008. [Online]. Available: https://doi.org/10.14778/1454159.1454211

[53] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[54] Yahoo! Research, "YCSB Core Workloads," Online: https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads, 2020, accessed: April 2020.

[55] A. C. De Melo, "The New Linux'perf'Tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[56] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, " Workload Analysis of a Large-scale Key-value Store ," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 53–64, Jun 2012.

[57] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," in *In Proc. of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM'99*, vol. 1. IEEE, Mar 1999, pp. 126–134 vol.1.

[58] Twitter, Inc., "Twemproxy (nutcracker)," Online: https://github.com/twitter/twemproxy, 2020, accessed: April 2020.

[59] C. Gormley and Z. Tong, *Elasticsearch: The Definitive Guide: a Distributed Real-time Search and Analytics Engine*. " O'Reilly Media, Inc.", 2015.

[60] A. Białecki, R. Muir, G. Ingersoll, and L. Imagination, "Apache Lucene 4," in *SIGIR 2012 workshop on open source information retrieval*, 2012, p. 17.

[61] ElasticSearch, "EsRally track," Online: https://github.com/elastic/rally-tracks,2020,access, 2020, accessed: April 2020.

[62] A. Forencich, V. Kamchevska, N. Dupuis, B. G. Lee, C. W. Baks, G. Papen, and L. Schares, "A Dynamically-Reconfigurable Burst-Mode Link Using a Nanosecond Photonic Switch," *Journal of Lightwave Technology*, vol. 38, no. 6, pp. 1330–1340, 2020.

[63] N. Dupuis, F. Doany, R. A. Budd, L. Schares, C. W. Baks, D. M. Kuchta, T. Hirokawa, and B. G. Lee, "A $4\times$ 4 Electrooptic Silicon Photonic Switch Fabric With Net Neutral Insertion Loss," *Journal of Lightwave Technology*, vol. 38, no. 2, pp. 178–184, 2020.

[64] N. Dupuis, J. E. Proesel, N. Boyer, H. Ainspan, C. W. Baks, F. Doany, E. Cyr, and B. G. Lee, "An $8\times$ 8 Silicon Photonic Switch Module with Nanosecond-Scale Reconfigurability," in *Optical Fiber Communication Conference*. Optical Society of America, 2020, pp. Th4A–6.

[65] W. M. Mellette, R. Das, Y. Guo, R. McGuinness, A. C. Snoeren, and G. Porter, "Expanding Across Time to Deliver Bandwidth Efficiency and Low Latency," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 1–18.

[66] M. Technologies, " ConnectX®-6 Single/Dual-Port Adapter supporting 200Gb/s with VPI ," Online: https://www.mellanox.com/products/infiniband-adapters/connectx-6, accessed: February 2020.

[67] P. Maniotis, L. Schares, B. Lee, M. Taubenblatt, and D. Kuchta, "Scaling HPC Networks with Co-packaged Optics," in *Optical Fiber Communication Conference*. Optical Society of America, 2020, pp. T3K–7.

[68] 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.

[69] 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.

[70] S. Legtchenko, N. Chen, D. Cletheroe, A. Rowstron, H. Williams, and X. Zhao, " XFabric: A Reconfigurable In-Rack Network for Rack-Scale Computers ." Santa Clara, CA: USENIX Association, March 2016, p. 15–29. [Online]. Available: https://www.microsoft.com/en-us/research/publication/xfabric-a-reconfigurable-in-rack-network-for-rack-scale-computers/