Chaosity: Understanding Contemporary NUMA-architectures

Hamish Nicholson[1,2⋆](✉), Andreea Nica[1], Aunn Raza[1], Viktor Sanca[1], and Anastasia Ailamaki[1,2⋆]

[1] EPFL, Data-Intensive Applications and Systems Lab, Lausanne, Switzerland
`first.last@epfl.ch`
[2] Google, Sunnyvale, USA

Abstract. Modern hardware is increasingly complex, requiring increasing effort to understand in order to carefully engineer systems for optimal performance and effective utilization. Moreover, established design principles and assumptions are not portable to modern hardware because: 1) Non-Uniform Memory Access (NUMA) architectures are becoming increasingly complex and diverse across CPU vendors; Chiplet-based architecture provides hierarchical NUMA instead of flat-NUMA topology, while heterogeneous compute cores (e.g., Apple Silicon) and on-chip accelerators (e.g., Intel sapphire rapids) are also normalized in materializing the vision for workload- and requirement-specific compute scheduling. 2) Increasing IO bandwidth (e.g., arrays of NVMe drives approaching memory bandwidth) is a double-edged sword; having high-bandwidth IO can interfere with the concurrent memory access bandwidth as the IO target is also memory; hence IO itself consumes memory bandwidth. 3) Interference modeling is becoming more complex in modern hierarchical NUMA and on-chip heterogeneous architectures due to topology obliviousness. Therefore, systems designs need to be hardware topology-aware, which requires understanding the bottlenecks and data flow characteristics, and then adapting scheduling over the given hardware topology.

Modern hardware promises performance by providing powerful and complex yet non-intuitive computing models which require tuning specifically for target hardware or risk under-utilizing the hardware. Therefore, system designers need to understand, carefully engineer, and adapt to the target hardware to avoid unnecessarily hitting bottlenecks in the hardware topology. In this paper, we propose the Chaosity framework, which enables system designers to systematically analyze, benchmark, and understand complex system topologies, their bandwidth characteristics, and interference of effects of data access paths, including memory and PCIe-based IO. Chaosity aims to provide critical insights into system designs and workload schedulers for modern NUMA hierarchies.

Keywords: NUMA · Data Access · IO · NVMe · Throughput · Interference.

⋆ Work done entirely at EPFL.

1 Introduction

Post Moore's law era, hardware designs have tended to scale horizontally, scaling compute via partitioning and packing more computational units in a single chip. In addition to multi-core processors, NUMA (non-uniform memory access) sockets add another layer of compute partitioning, creating an almost distributed yet coherent and shared-everything system in a single scale-up server. Initially, NUMA hardware had non-uniformity, but all CPUs were treated as homogeneous processors, while accelerators were treated as co-processors for each NUMA node. Essentially, NUMA nodes were arranged as siblings in the hardware stack, having their own memory hierarchy, including PCIe-attached storage and a set of co-processors optionally.

Advancements in the hardware landscape challenge traditional system designs to avail the performance and efficiency offering from the modern hardware and maintain performance standards [11]. The fundamental changes are:

1. Hierarchical NUMA in chiplet-based architectures, even in single-socket machines in mainstream CPUs offered by AMD [12] and Intel [13].
2. IO-bandwidth competing with memory bandwidth. Moreover, co-processors and other sibling PCIe-attached devices may compete in bandwidth utilization by directly consuming from the devices (e.g., GPU reading from NVMe without involving CPUs), compared to previously strict uni-directional storage hierarchy [15].
3. Heterogeneous compute-on-chip is becoming the norm rather than a niche. For example, Apple Silicon [8] and Intel [4] consumer-grade chips have different types of compute units within the same chip, specialized for a range of workloads, and Intel Sapphire Rapids offer on-chip, off-core accelerators. This introduces additional heterogeneity in compute scheduling and data routing [13].

Consequentially, tuning systems with traditional but usurped design principles in mind, including but not limited to NUMA-aware partitioning and caching across memory hierarchy, may result not only in a lack of speedup but also in performance regressions. Modern hardware promises increased performance and scalability when tuned to the expected software design. This is because, previously, hardware across vendors and generations was mostly homogeneous in design principles. For example, a multi-socket, multi-core machine was assumed to have three layers of caches for each processor, the first two levels private to each core and a shared last-level cache, and a high-speed coherent link between all NUMA nodes. In general, each newer processor generation added new features but was mostly transparent to the user regarding software design, therefore, was compatible with existing NUMA-aware systems. However, with modern hardware, the hardware topology is not homogeneous across vendors: AMD EPYC has a heterogeneous chiplet-based architecture [11], while Intel Sapphire Rapids [13] offers on-chip accelerators and even high-bandwidth memory in certain models. Further, depending on the specific hardware, the NUMA topology may be hierarchical in the case of chiplets, creating a tree of NUMA nodes.

This results in the fundamental invalidation of software design principles and assumptions. For example, the used-to-be flat NUMA-aware partitioning would suffer from interference in remote accesses in a hierarchical NUMA topology. Such trends require hardware-software co-design, but it requires either software designed especially for specific hardware only, which is hardly the case, or adaptive across hardware. In any case, the first step in designing a system that could efficiently utilize the powerful features of modern hardware is *understanding the hardware*. Make no mistake: understanding hardware does not mean studying it at the silicon level, but understanding the hardware topology and capabilities, the data-flow paths, and associated characteristics. For example, understanding the bandwidth difference between memory and disk is critical in designing caching policies [14,15]. In another case, an algorithm design would differ based on the availability of coherent versus non-coherent interconnect between a CPU and an accelerator (e.g., GPUs).

Therefore, we propose Chaosity, a framework for systematically understanding hardware topology, bandwidth characteristics for memory and PCIe-based IO, and modeling interference between non-partitioned memory operations. This is a first step towards automatic benchmarking and bootstrapping critical and actionable insights required for a systems engineer to understand and for an adaptive system to tune itself for specific hardware. The rest of the paper is organized as follows:

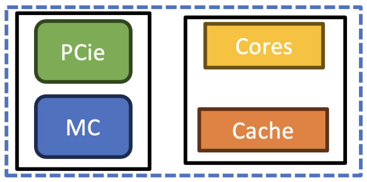
- Section 2 highlights the increasing heterogeneity in the hardware landscape, which motivates the importance and impact of systematic analysis,
- Section 3 focuses on the initial design of an automated non-uniformity benchmark,
- Sections 4 and 5 presents experimental results that demonstrate the need for a chaos-aware heterogeneous platform on two such architecture configurations,
- Section 6 discusses the takeaways, implications and provides future directions for this work
- Section 7 concludes the vision of Chaosity.

2 Motivation: Rising Entropy in the Hardware Landscape

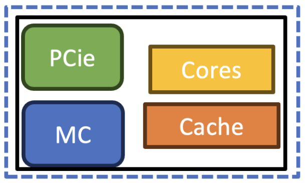
The advancements in computer architecture change the system landscape and the opportunities for hardware-software co-design. Figure 1 shows the evolution of recent mainstream CPU topologies. Besides the well-known transition from single-core to multi-core architectures, the chip shrinkage has allowed integrating components from a single Northbridge/memory controller hub (Figure 1a) into a single chip die (Figure 1b), alleviating the bottleneck of data transmission and introducing NUMA with on-socket memory controllers.

2.1 Hierarchical NUMA

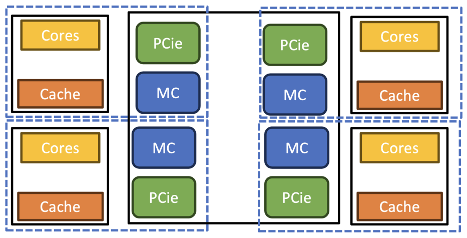
The continuation of chip downsizing has led to the post-Moore law era, leading to challenges in CPU scalability where vendors are increasingly adopting chiplet



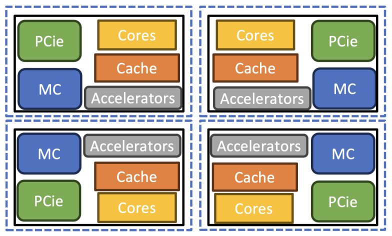
(a) Memory controller and IO on north bridge



(b) Integrated memory controller style



(c) AMD EPYC Milan style



(d) Intel Sapphire Rapids style

Fig. 1: Evolution of recent CPU topologies. Solid black boxes represent chips/chiplets, and dashed blue boxes represent NUMA regions

(multi-chip module) designs (Figure 1c) [12,13]. This changes the traditional monolithic CPU design when non-uniform memory access (NUMA) resulted from multi-socket CPU servers. Chiplets introduce additional NUMA regions even inside a single socket in a hierarchical fashion, increasing the complexity of main memory access paths. Though chiplet-based CPUs are not the first CPUs to expose multiple NUMA regions within a single socket, they are more widely adopted than previous commercial offerings, such as Intel's Xeon Phi Knights Landing. [19,23]). AMD has been using chiplet designs since the EPYC Naples generation [12], while Intel recently moved to chiplets for many of their server CPUs in the Intel Sapphire Rapids [4] generation. The chiplet designs from Intel and AMD allow exposing the hierarchical NUMA regions to the operating system, e.g., 8 NUMA nodes in a 2-socket server.

2.2 On-chip Heterogeneity

The complexities do not end at hierarchical NUMA. To add to memory and data access path complexity, even the compute units may not be uniform. Specialized accelerators and non-uniformity introduce differences in throughput and memory access (Figure 1d).

The benefit of having specialized or heterogeneous cores on a single chip is clear to all hardware vendors, and thereby, in the last few years, we have seen consumer-grade CPUs like Apple M1/M2 silicon and 12th generation Intel Core desktop processors packaging performance and efficiency cores in a multi-core chip, scaling the non-uniformity axis to heterogeneous compute units. Further, besides the disparity between different cores, both consumer- and server-grade hardware introduced specialized on-chip off-core accelerator components optimized for specialized or specific workloads, such as neural processing. To list a few, Apple Silicon has an accompanying neural engine for machine learning (ML) workloads. Intel Sapphire Rapids CPU comes with specialized tile registers and matrix multiplication intrinsic (AMX) for ML, data encryption and compression accelerators (QAT), and data streaming accelerator (DSA). Recently, AMD also announced MI300 APUs (accelerated processing units) composed of modular chiplets. The AMD MI300 APU will offer a combination of CPU (Zen4) or GPU (CDNA3) chiplets and on-chip high-bandwidth memory [22]. Further, Nvidia Grace Hopper Superchips tightly integrate an ARM-based CPU and an NVIDIA GPU chip with a fast inter-chip NVLINK interconnect [17].

2.3 Data Highways: Interconnect

Interconnects play a significant role in data access and movement across complex topologies as careful use of the limited available bandwidth is crucial for efficiency [21,6,15]. PCIe interconnects in complex topologies (Figure 2) represent a shared resource between CPUs, accelerators, and IO devices.

An added complexity ensues with the addition of per-hierarchical-NUMA PCIe controllers and, for example, fast NVMe drives, which are in aggregate on par with the available main-memory bandwidth, interfering and contending with the even more complex memory access path. IO predominantly uses direct memory access (DMA) to transfer data between devices and main memory. To perform a DMA transfer to read data from an IO device, the CPU submits a request to the device and then the device's DMA engine is responsible for transferring the data directly to main memory.[3] The CPU can then read the data from memory using load instructions. The processing for writing data to an IO device inverts the order of operations. Still, there is no uniform design approach for interconnects, as Apple Silicon has a unified memory architecture for their CPU, GPU, and neural engines. Overall, the location of the IO device and the access path complexity requires careful coordination and placement.

2.4 Systems with Complex Data Access

Traditionally, the hardware topologies were mostly homogeneous and standard across the vendors, and therefore, the underlying hardware performance was more predictable and understandable by the on-paper specifications. The system designers would have to consider only a few metrics, including but not limited to memory bandwidth, CPU interconnect bandwidth, and PCIe or device bandwidth. Most scalable applications were designed with NUMA-aware partitioning and cache-aware algorithms, catering to both CPU caches when reading/writing

[3] Intel Xeon CPUs feature Data Direct IO (DDIO), which transparently allows PCIe devices to read/write directly to last-level caches initially bypassing DRAM. [1]

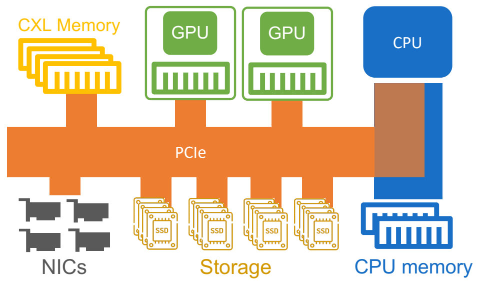


Fig. 2: Interconnects contribute to complex heterogeneous topologies

from memory to CPU and buffer pools when reading/writing from disk to memory. However, with ever-increasing heterogeneous and complex hardware, mere NUMA-partitioning and cache-aware algorithms do not fully utilize the underlying hardware but may result in performance regressions in some cases. For instance, Nicholson et al.[14,15] concluded that with high-bandwidth storage, simple frequency- or recency-based caching is sub-optimal and requires proportional caching to utilize the high-bandwidth storage fully. Moreover, not only do the system designs have to cater to fully utilizing the underlying hardware capabilities, but they also need to account for the interference domain. For instance, Raza et al.[20,21] partitioned latency-sensitive and bandwidth-intensive workloads across NUMA-boundaries to alleviate interference in the memory hierarchy but was limited by the interconnect; however, with chiplet architectures, the system could have partitioned the interfering workloads across chiplet boundaries.

Putting everything together, increasingly high hardware complexity opens up tuning opportunities to profit from and avoid performance regressions. Such opportunities may be trivial and intuitive: partitioning workload across NUMA boundaries, minimizing data movements, etc., or, non-intuitive based on the underlying hardware characteristics: de-prioritizing data locality in favor of partitioning workload based on interference, or staging/buffering IO in system memory for granular IO from PCIe-attached devices like GPUs [15]. Still, a given workload will have a combination of memory, computational, and data movement requirements that might have a desirable particular hardware constellation. This motivates a systematic study of the complexity and the chaos ensuing, not only for a given platform and workload but for any future change in platform or workload.

3 Chaosity Framework - Understand thy Hardware

Characterizing non-uniformity and interference is highly challenging. While benchmarks traditionally target a specific and often limited set of parameters, previously described complex hardware and data movement interactions instead exacerbate the need for a holistic benchmark, such as memory-IO interference or CPU-GPU interference with shared memory. Still, the design space of interference micro-benchmarks is vast, motivating for a framework that composes micro-benchmarks that best represent a workload to analyze the complex effects of the underlying system and hardware.

To this extent, as good examples of characterizing system-crucial characteristics, we employ fio [3] and STREAM [10]. fio benchmarks persistent disk performance (IO), and STREAM measures sustained memory bandwidth. This way, we aim to analyze and understand the interference between the two subsystems. Our design allows easy incorporation of existing workloads or ones written from scratch. For example, rather than measuring memory-IO interference only, CPU cache interference would allow for finer-granularity benchmark experiments.

Benchmark categorization and selection The first step in profiling any hardware is the categorization of benchmarks, that is, the target metric or characteristics to be measured. The benchmark category defines the benchmarks that will be executed across independent and shared configurations. Each benchmark category measures the specific target property or metric of the hardware under test. Currently, benchmarking categories include memory and PCIe-based storage bandwidth.

The second step is defining which benchmark to use, either a standard off-the-shelf benchmark or a customized benchmark. The invariant in one or more benchmark selections is the target metric. Chaosity provides a default standard benchmark for each category. However, one can add or replace the benchmark with another standard or custom benchmark. For instance, by default, Chaosity utilizes STREAM [10] for profiling memory bandwidth and fio [3] for profiling storage bandwidth.

Component and topology discovery Chaosity begins profiling hardware under test by first discovering the available components/devices and their topology and memory model. For example, detecting the number of available cores, the number of hyper-threads per physical core, NUMA nodes, sockets, and connected devices, including but not limited to NVMe storage and GPU devices. Chaosity leverages *hwloc* and Linux *numactl* utilities to discover the hardware topology [5]; Further, Chaosity queries the underlying hardware properties, such as detecting if the devices have a unified memory and if the shared memory is coherent across devices, CPU cache-line size, etc. Chaosity also detects the availability of pre-defined specialized accelerators, such as on-chip accelerators in Intel Sapphire Rapids.

Discovering hardware components, their topology, and associated properties is critical in profiling and benchmarking as it defines the interaction between different components and the expected behaviors.

Executing independent benchmarks. After benchmark categorization, definition, and hardware discovery, Chaosity begins the profiling defined by each benchmark category. The benchmark utilizes the topology information and starts by profiling the minimum unit of each type of compute and, from thereon, profiles the combination of components in the hierarchy. Chaosity needs to profile hierarchy for all combinations to detect any non-uniformity, asymmetry, or some-times, even hardware defects or misconfigurations. In the following, we detail the memory and storage bandwidth benchmarking.

Benchmarking memory bandwidth. Analyzing memory bandwidth starts with the default unit of compute, that is, single-thread in CPU, and then proceeds by analyzing bandwidth of single-NUMA, single-socket, and then all CPUs. Then, the system profiles remote interactions, which in the case of memory, means accessing remote memory for each unit, starting from bandwidth for accessing the memory of different NUMA nodes within the same socket and then similarly for remote sockets.

Benchmarking storage bandwidth. Analyzing storage bandwidth proceeds similarly to analyzing memory bandwidths. However, it adds an additional basic unit, the number of drives attached locally to each NUMA node, to analyze the bandwidth scalability across combinations of the connected drives within the same and other NUMA nodes.

Memory-storage interference modeling. A shared memory subsystem introduces competition in data accesses and hence, causes interference. This interference occurs at all levels of the memory hierarchy, including competing cache lines, load requests, and memory bandwidth itself. For now, we target and model memory bandwidth interference, which arises when accessing both CPU memory and storage or remote memory over PCIe. The bandwidth interference arises as memory access goes through the same memory controller in most processor architectures. Hence, a memory controller can only process a certain amount of data simultaneously, prioritizing one over another.

Chaosity profiles and models the interference by scheduling independent memory and storage bandwidth benchmarks concurrently. It profiles the interference by collocating and isolating the compute unit and the read drive set using topology information. In doing so, it models the interference when both accesses are issued from the same controller or are routed through a different controller. Ideally, when co-scheduled, the total bandwidth (memory + storage) should be equal to the max of either; however, when scheduled across NUMA boundaries, should not interfere with each other as the PCIe root complex should be directly accessible from the requesting memory controller. However, it is hardly the ideal case due to the hidden complexities of hardware design, and therefore, it is crucial to profile and understand the bandwidth degradation in all cases, guiding system designers to account for and schedule workloads accordingly.

4 Heterogeneous Compute Units – Apple M1 Pro Silicon

In this section, we use Chaosity to test Apple M1 Pro silicon, explicitly targeting the unified memory bandwidth across heterogeneous CPU cores and GPU and analyzing the maximum memory bandwidth when executed in isolation and the degradation due to interference when all compute units compete for memory bandwidth.

Hardware. Apple Macbook Pro 2021 running macOS 12.5.1 with a M1 Pro processor (Model identifier: MacBookPro18.3, Model number: Z15G002BDSM/A), having 10 CPU cores (eight performance and two efficiency cores), 16 GPU cores, and 16 neural-engine cores, with a total of 32GB LPDDR5 memory and a 512GB NVMe SSD.

Benchmark. On the CPU, memory bandwidth tests utilize the STREAM Triad benchmark [10]. We report the average bandwidth over 100 iterations for STREAM, not including the first iteration. We compile STREAM with Clang 13.0.1 with the `-O2 -fopenmp -DSTREAM_ARRAY_SIZE=80000000` compiler flags. In this configuration, each array element is an 8-byte double. While OpenMP is used to control the number of STREAM benchmark threads, it cannot bind threads to cores as on Linux. This is because macOS has no underlying API to pin threads. To run STREAM on efficiency cores, we use the `taskpolicy` system utility to launch STREAM with the `PRIO_DARWIN_BG` scheduling priority [2]. On the GPU, memory bandwidth tests use a variation of the `bw_benchmark` from [9]; this benchmark performs a multiply-add on 3 input [8192 $\times$ 8192] matrices of 32 bit floats and stores the output in another matrix of the same type, using a total of 1 GiB of memory. For NVMe bandwidth tests, we use fio [3] to perform sequential reads. Our fio configuration for macOS uses the posixaio engine, a 1MB blocksize, `O_DIRECT`, and is time based to run for 30 seconds.

4.1 Interference in Unified Memory

Apple Silicon has a unified memory across all types of compute units, including performance and efficiency CPU cores, GPU cores, and neural-engine cores. Unified memory offers coherent access across heterogeneous consumers, that is, compute units or networks and other devices in some cases. Apple silicon is different from traditional CPU-GPU unified memory (like Nvidia's Unified Memory) in the sense that all types of compute cores are at the same level, and the last-level-cache is shared across all heterogeneous cores, which in our view, simplifies the cache coherency implementation in hardware. However, there is no free lunch. In the general case, not all compute devices will be running data-intensive operations. Still, for high-performance or analytical data processing tasks, all devices will execute data-intensive tasks and, thereby, require the maximum possible bandwidth to underlying unified memory.

Table 1 shows the experimental results when running memory benchmarks on Apple M1 silicon in different configurations. For standalone CPU baselines, a single performance core can consume a maximum memory bandwidth of 75 GB/s, while a single efficiency core can only achieve maximum memory bandwidth of 11.5 GB/s. Whereas, utilizing all eight performance cores only, we get 128 GB/s while using both efficiency cores only, we get 15 GB/s, and utilizing all CPU cores, that is, ten cores (eight performance and two efficiency), we get a maximum memory bandwidth of 138 GB/s. In the case of GPU, when benchmarking in GPU-only mode, the benchmark achieves 176 GB/s, and to the best of our knowledge, there is no way of scheduling and affinitizing workload on the neural engine; hence, it is not included in the scope of this study.

From the baselines described above, which do not have any conflicting or interfering workload, it is clear that GPU-cores have access to 27% and 37%

Scheduling mode	Compute units	Memory bandwidths (GB/S)
CPU-only	1 Efficiency core	11.7
	2 Efficiency cores	14.8
	1 Performance core	75.2
	8 Performance cores	128.7
	All cores (8P + 2E)	138.3
GPU-only	All 16 GPU cores	176.3
CPU-GPU	8 P-CPU (w/ 16 GPU)	59.8
	16 GPU (w/ 8 P-CPU)	118.9
	10 CPU (w/ 16 GPU)	60.8
	16 GPU (w/ 10 CPU)	115.0

Table 1: Memory Bandwidth Analysis of Apple M1 Pro

more memory bandwidth compared to all CPU cores and all of the performance CPU cores.

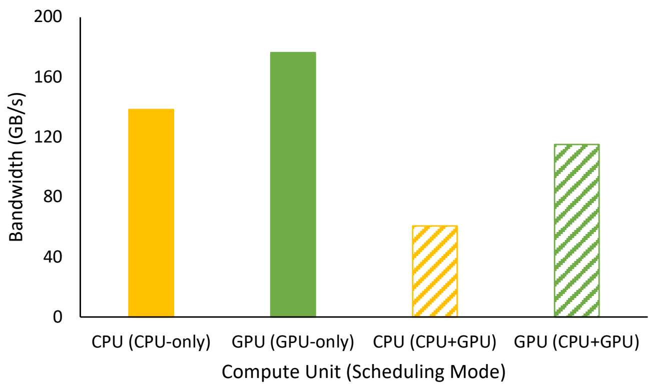


Fig. 3: Memory-storage bandwidth interference in Apple M1 Pro

Unified memory shares the memory access across all compute devices, hence sharing the bandwidth accordingly. We study bandwidth interference and priority by running parallel independent benchmarks on both CPU and GPU. As the benchmarks are executed independently the bandwidth interference arises purely from resources competing for accessing memory, but not from accessing objects in memory shared by the benchmarks. Figure 3 plots the bandwidth degradation when both efficiency and performance CPU cores are used concurrently with the GPU. Table 1 in addition shows results for just the performance CPU cores.

We observe that GPU still gets priority over CPU in bandwidth allocation. The GPU memory bandwidth drops 35% and 32% when running using only CPU performance and all CPU cores, respectively. Whereas CPU memory bandwidth degrades by 54% when using only performance cores and 56% when using all cores. In both cases, the sum of CPU and GPU memory bandwidths are nearly equal to the bandwidth the GPU observes when running independently. This shows that to leverage the full memory bandwidth, the GPU must be used, but with the trade-off that less bandwidth will be available for the CPU cores.

5 Chiplet-based Server – AMD EPYC

In this section, we use Chaosity to test and analyze the AMD EPYC (Milan) server processors having a chiplet-based architecture. AMD EPYC is a representative of modern hardware which has hierarchical NUMA, that is, chiplets. Additionally, it provides enough PCIe 4.0 lanes to saturate more than half of the memory bandwidth for each chiplet.

Hardware. All experiments were conducted on a server with a 2x24-core AMD EPYC 7413 processor, having two threads per core, totaling 96 threads and 256 GB of DRAM. Each CPU socket has 16 Corsair MP600 Pro NVMe drives, each using 4 PCIe 4.0 lanes. The manufacturer-specified maximum read bandwidth of each drive is 7 GB/s [7]. At measurement time, two drives failed; therefore, NUMA nodes 0 and 4 have 3 NVMe drives each, while all other NUMA nodes have four drives each. AMD EPYC has 128 PCIe 4.0 lanes per socket, theoretically having a total bandwidth of 256 GiB/s, whereas the main CPU can independently achieve an aggregate bandwidth of 128 GiB/s per socket. In a two-socket configuration, 48 lanes of PCIe are used on each chip for the inter-socket interconnect. The remaining lanes are available for other PCIe devices.

Our server is running Ubuntu 20.04 with Linux Kernel 5.4.

Benchmark. Memory bandwidth tests utilize the STREAM Triad benchmark [10]. STREAM is compiled using GCC 9.4 with the `-O2 -fopenmp -DSTREAM_ARRAY_SIZE=100000000 -mcmodel=medium` compiler flags. In this configuration, each array element is an 8-byte double. Numactl is used to set the NUMA nodes STREAM will execute on as well as to bind the memory used by STREAM to specific, possibly different, NUMA nodes. For NVMe bandwidth tests, we use fio [3] 3.32. Our default fio benchmark is a sequential access benchmark that uses the io_uring engine, a 1MB blocksize, `O_DIRECT`, and is time-based to run for 30 seconds.

5.1 Hierarchical NUMA

In the following analysis, we employ Chaosity to analyze memory and storage bandwidth in AMD EPYC Milan architecture and model the interference between the two. **Memory Bandwidth.** Figure 4 shows experimental results of measuring memory bandwidth grouped by the CPU cores and target memory nodes. Using numactl, STREAM is CPU bound to the NUMA node on the x-axis

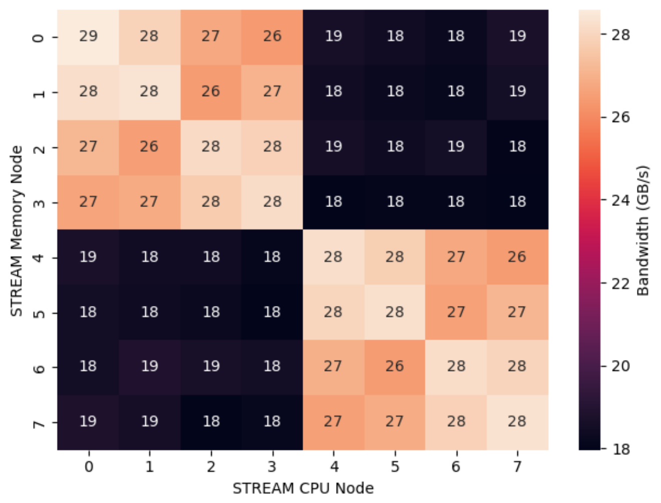


Fig. 4: Memory bandwidth in AMD EPYC – Each NUMA node accesses the memory of target NUMA node with STREAM TRIAD benchmark

and memory bound to the NUMA node on the y-axis. Each number represents an independent run; hence, no interference or contention in accessing local or remote memory. The memory bandwidth within the socket is similar while accessing the memory of a remote socket is 33% slower, regardless of whichever chiplet in the socket itself.

Figure 5 shows the results of analyzing the maximum storage bandwidth when all cores of the NUMA node read from all drives of the target node. The interesting thing to observe here is that the average maximum bandwidth substantially degrades when all the experiments where NUMA nodes of the first socket access the NVMe drives on the second socket. However, this is not the case for the opposite: NUMA nodes of the second socket accessing NVMe drives connected to the first socket.

To further elaborate on this behavior, figure 6 shows the maximum storage bandwidth achieved by each NUMA node for all combinations of NVMe drives. The throughput degradation observed in figure 5 is shown when the first socket accesses all drives from individual NUMA nodes of the second socket. However, it is compensated when read from drives of multiple NUMA nodes of the second socket. Secondly, for both sockets, when accessing from NVMe drives of the first and last NUMA node of the remote socket, the throughput degrades (fio-node 4,5,6,7 accessing data from [0,3] and fio-node 0,1,2,3 accessing data from [4,7]). To the best of our knowledge, the reason for this behavior is unknown and may be a fault in hardware or software configuration. However, this is one of the main benefits of Chaosity: targeting and identifying such unexpected and anomalous behaviors. Without Chaosity, one would have deployed a fully

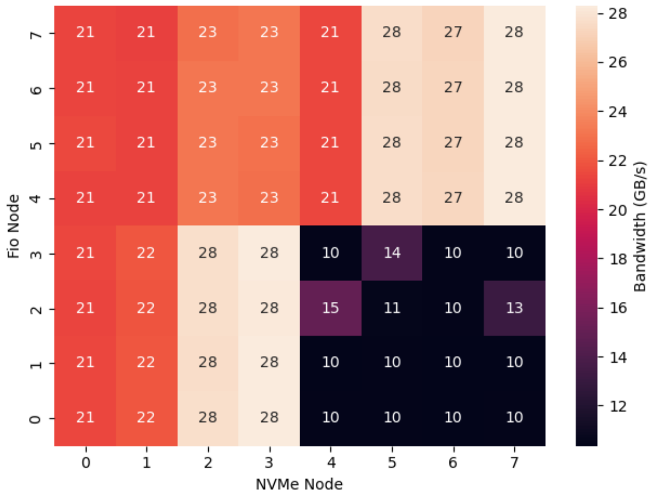


Fig. 5: Storage bandwidth (GB/s) in AMD EPYC – All cores of NUMA node accesses all NVMe drives in target NUMA node

functional system and then spent time analyzing system performance regression while not knowing that the actual issues are not in the system design but in the hardware or hardware configuration.

5.2 Bandwidth Interference – Interconnect & Memory

PCIe data transfers also consume memory bandwidth when reading or writing from/to CPU memory. One such case is when reading or writing data from/to NVMe drives. This causes interference and, counter-intuitively, consumes the memory bandwidth, limiting the processors' data processing performance. In what follows, we analyze the interaction of PCIe bandwidth with memory bandwidth and provide insights for data-intensive processing.

Figure 7 plots the interaction between IO and memory bandwidth on a single NUMA node. Data is read from the NVMe drives using fio, while simultaneously, STREAM is run on the same NUMA node. As the number of NVMe drives read from increases, more memory bandwidth is used for IO, and we observe lower memory bandwidth consumed by STREAM. This exemplifies the point that IO bandwidth consumes memory bandwidth. This is increasingly relevant as storage bandwidths increase, resulting in memory bandwidth competition.

Figure 8 demonstrates competition for memory bandwidth between IO and a simple compute kernel in a single program. The compute kernel performs a summation over an array of integers, using one socket of the server. The black line shows the processing throughput when the array is entirely in memory. The blue line shows the throughput when the array is striped across an increasing number of NVMe drives on the socket. When the array is NVMe resident, it

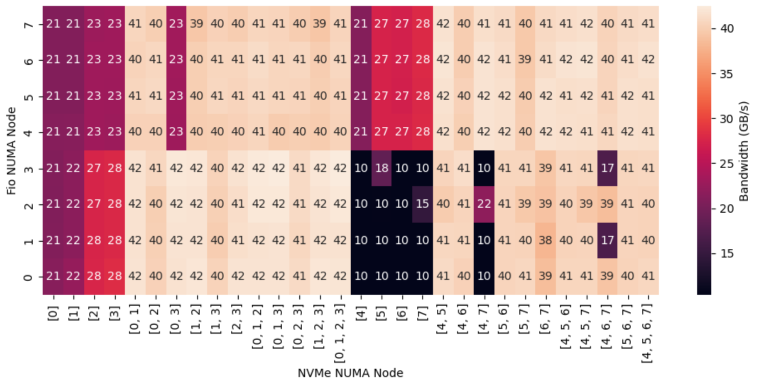


Fig. 6: Storage bandwidth (GB/s) in AMD EPYC – All cores of NUMA node accesses specified NVMe drives in target NUMA node

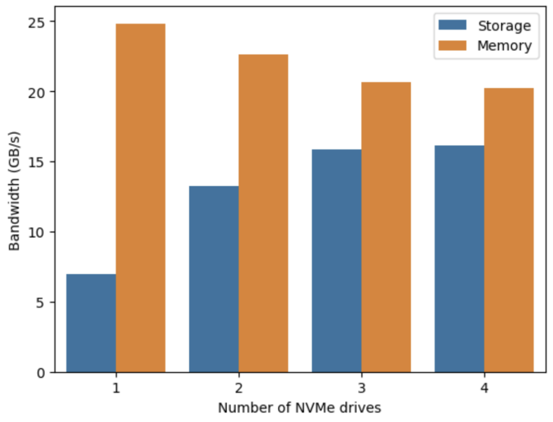


Fig. 7: Interference in read bandwidth between storage and memory

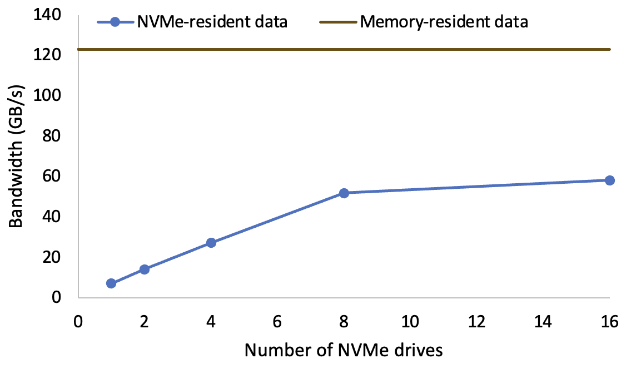


Fig. 8: CPU and IO compete for memory bandwidth within the same workload.

is asynchronously transferred in 2 MiB chunks into memory using io_uring, as the chunks arrive in memory they are consumed by the compute kernel so that the compute and NVMe transfers are overlapped. This experiment differs from simply using fio, as fio only transfers data from storage to memory and does not also load the transferred data again from memory to the CPU.

Operating on storage resident data consumes twice the memory bandwidth than operating on in-memory data. We observe that for the baseline memory-resident data, the processing throughput is 120 GiB/s, near the single socket memory bandwidth. When reading from storage, the number of drives, and hence the storage bandwidth, is the initial bottleneck. Whereas, with eight or more drives, the available storage bandwidth does not improve the processing throughput; as the transfers from storage consume memory bandwidth to write to memory and the CPU utilizes the remaining memory bandwidth to read the data from memory. At this point, the bottleneck has shifted to memory bandwidth.

6 Discussion

In sections 4 and 5, we presented experimental results which demonstrate some of the complexities of the modern hardware landscape that data-intensive systems developers must account for. Architectures like the Apple M1 can only fully utilize the memory bandwidth by using the GPU but at the cost of interfering with the CPU's memory bandwidth. The EPYC Milan architecture allows for tremendous PCIe bandwidth, which can be used for NVMe storage, but the PCIe/storage bandwidth cannot be fully utilized if the CPU also needs to transfer the data between its caches and memory as memory bandwidth becomes the bottleneck. Our results are only for two specific topologies. However, servers can be configured in many different ways, for example, with PCIe network interface cards (NIC) and accelerators. This enables additional data transfer paths such as storage to accelerator and storage to NIC transfers, which bypasses the main memory but still consume IO bandwidth [16,18].

The complex topology of modern servers, both due to varying CPU architectures and possible server configurations, severely increases the cognitive load on designers of data-intensive systems. System designers strive to maximize hardware utilization in order to minimize the cost and energy use of their systems. We expect two common approaches to achieve good utilization in the era of diverse hardware. First, organizations that both develop software and deploy on hardware they manage, such as large cloud companies, may evaluate multiple types of servers and settle on one or a small number to deploy and optimize for. Second, systems that need to achieve portable performance can adapt to the topology at run-time. Still, both approaches necessitate that system designers have a deep understanding of hardware.

Our long-term goal is to assist and enable system designers to understand hardware better. Through topology and interference-aware benchmarking, Chaosity enables the exploration of the limits of the hardware in more realistic scenarios than individual system-wide single-metric benchmarks. Chaosity can be a complimentary tool to software system-specific benchmarks, as it aims to reveal the characteristics of the hardware rather than the performance of a, potentially untuned, software system on new hardware. This can be especially helpful because discovering performance bottlenecks due to interference at the hardware level is time-consuming to discover through profiling alone. Chaosity synchronizes the hardware expectations and reality given the current configuration and may also detect misconfigurations and defects early rather than wasting time in debugging/profiling a full software system.

Future directions. We envision Chaosity to be integrated with automatic topology discovery and adaptive components in a system as an input provider, thereby assisting systems in adapting to increasingly complex underlying hardware. We aim to include support for more types of benchmarks in Chaosity, including but not limited to benchmarking latency profiles, on-chip, off-chip accelerators and devices connected via Compute Express Link (CXL). Further, we also plan to add a shared and private profiling database to compare different hardware characteristics across different hardware types, vendors, and generations. We will encourage vendors and third parties to publish and compare results against standard and non-trivial configurations.

7 Conclusion

Modern hardware requires co-optimizing hardware and software. However, modern servers are becoming more diverse and heterogeneous. This complexity is a result of both CPUs that are scaling silicon horizontally and may also contain heterogeneous compute, as well as the increasing use of high-bandwidth IO devices and accelerators attached via an interconnect like PCIe. The diversity of server topologies will only continue to grow as novel CPUs come to market and new interconnects such as CXL enable new types of devices. Collectively, this poses new challenges for efficient and high-performance system designs.

This paper proposes an initial vision for the Chaosity framework, which assists system designers and developers in understanding the target hardware topology and associated performance characteristics. Further, hardware or software configurations are prone to misconfigurations, given the complex hardware topologies and systems designs. Chaosity will assist in detecting such problems in the early stages of hardware or software deployments by providing insights into expected hardware performance.

Acknowledgements. We would like to thank the reviewers for their valuable feedback. This work was partially funded by SNSF project "Efficient Realtime Analytics on General-Purpose GPUs" subside no. 200021_178894/1.

References

1. Alian, M., Yuan, Y., Zhang, J., Wang, R., Jung, M., Kim, N.S.: Data Direct I/O Characterization for Future I/O System Exploration. In: 2020 IEEE International

Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 160–169 (Aug 2020). https://doi.org/10.1109/ISPASS48437.2020.00031

2. Apple Inc.: macOS Taskpolicy Man Page, https://github.com/apple-oss-distributions/system_cmds/blob/559f661c5687f7828307cb3b1026a45f849243c6/taskpolicy.tproj/taskpolicy.8
3. Axboe, J.: Flexible I/O Tester (2022), https://github.com/axboe/fio
4. Biswas, A.: Sapphire rapids. In: 2021 IEEE Hot Chips 33 Symposium (HCS). pp. 1–22 (2021). https://doi.org/10.1109/HCS52781.2021.9566865
5. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: A generic framework for managing hardware affinities in hpc applications. In: 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. pp. 180–186 (2010). https://doi.org/10.1109/PDP.2010.67
6. Chrysogelos, P.: Efficient Analytical Query Processing on CPU-GPU Hardware Platforms. Ph.D. thesis, EPFL, Lausanne (2022). https://doi.org/10.5075/epfl-thesis-8068, http://infoscience.epfl.ch/record/296204
7. Corsair MP600 PRO 2TB M.2 NVMe PCIe Gen. 4 x4 SSD, https://www.corsair.com/us/en/p/data-storage/cssd-f2000gbmp600pro/mp600-pro-2tb-m-2-nvme-pcie-gen-4-x4-ssd-cssd-f2000gbmp600pro
8. Kenyon, C., Capano, C.: Apple silicon performance in scientific computing. In: IEEE High Performance Extreme Computing Conference, HPEC 2022, Waltham, MA, USA, September 19-23, 2022. pp. 1–10. IEEE (2022). https://doi.org/10.1109/HPEC55821.2022.9926315
9. Liu, T.: Tf-metal-experiments (2021), https://github.com/tlkh/tf-metal-experiments
10. McCalpin, J.D.: 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)
11. Naffziger, S., Beck, N., Burd, T., Lepak, K., Loh, G.H., Subramony, M., White, S.: Pioneering chiplet technology and design for the AMD Epyc™ and Ryzen™ processor families: Industrial product. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). pp. 57–70. IEEE (2021). https://doi.org/10.1109/ISCA52012.2021.00014
12. Naffziger, S., Lepak, K., Paraschou, M., Subramony, M.: AMD chiplet architecture for high-performance server and desktop products. In: 2020 IEEE International Solid-State Circuits Conference-(ISSCC). pp. 44–45. IEEE (2020). https://doi.org/10.1109/ISSCC19947.2020.9063103
13. Nassif, N., Munch, A.O., Molnar, C.L., Pasdast, G., Lyer, S.V., Yang, Z., Mendoza, O., Huddart, M., Venkataraman, S., Kandula, S., Marom, R., Kern, A.M., Bowhill, B., Mulvihill, D.R., Nimmagadda, S., Kalidindi, V., Krause, J., Haq, M.M., Sharma, R., Duda, K.: Sapphire rapids: The next-generation Intel Xeon scalable processor. In: 2022 IEEE International Solid- State Circuits Conference (ISSCC). vol. 65, pp. 44–46 (2022). https://doi.org/10.1109/ISSCC42614.2022.9731107
14. Nicholson, H., Chrysogelos, P., Ailamaki, A.: Hpcache: Memory-efficient OLAP through proportional caching. In: Blanas, S., May, N. (eds.) International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022. pp. 7:1–7:9. ACM (2022). https://doi.org/10.1145/3533737.3535100
15. Nicholson, H., Raza, A., Chrysogelos, P., Ailamaki, A.: Hetcache: Synergising NVMe storage and GPU acceleration for memory-efficient analytics. In: 13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The

Netherlands, January 8-11, 2023. www.cidrdb.org (2023), https://www.cidrdb.org/cidr2023/papers/p84-nicholson.pdf

16. NVIDIA: GPUDirect Storage: A direct path between storage and gpu memory (Aug 2019), https://developer.nvidia.com/blog/gpudirect-storage/
17. NVIDIA: NVIDIA Grace Hopper Superchip Architecture Whitepaper (2023), https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper
18. NVMe Standard: NVM Express RDMA Transport Specification (2022), https://nvmexpress.org/wp-content/uploads/NVM-Express-RDMA-Transport-Specification-1.0b-2022.10.04-Ratified.pdf
19. Ramos, S., Hoefler, T.: Capability models for manycore memory systems: A case-study with Xeon Phi KNL. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 297–306 (2017). https://doi.org/10.1109/IPDPS.2017.30
20. Raza, A., Chrysogelos, P., Anadiotis, A.G., Ailamaki, A.: Adaptive HTAP through elastic resource scheduling. In: Maier, D., Pottinger, R., Doan, A., Tan, W., Alawini, A., Ngo, H.Q. (eds.) Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020. pp. 2043–2054. ACM (2020). https://doi.org/10.1145/3318464.3389783
21. Raza, A., Chrysogelos, P., Sioulas, P., Indjic, V., Anadiotis, A.G., Ailamaki, A.: Gpu-accelerated data management under the test of time. In: 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings. www.cidrdb.org (2020), http://cidrdb.org/cidr2020/papers/p18-raza-cidr20.pdf
22. Su, L.: AMD CES 2023 keynote (2023), http://www.ces.tech/sessions-events/keynotes.aspx
23. Williams, S., Ionkov, L., Lang, M.: Numa distance for heterogeneous memory. In: Proceedings of the Workshop on Memory Centric Programming for HPC. p. 30–34. MCHPC'17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3145617.3145620