CHARM: Chiplet Heterogeneity-Aware Runtime Mapping System

Alessandro Fogli
Imperial College London
London, United Kingdom
a.fogli18@imperial.ac.uk

Bo Zhao
Aalto University
Espoo, Finland
bo.zhao@aalto.fi

Peter Pietzuch
Imperial College London
London, United Kingdom
prp@imperial.ac.uk

Jana Giceva
TU Munich
Munich, Germany
jana.giceva@in.tum.de

Abstract

The growing disparity between CPU core counts and available memory bandwidth has intensified memory contention in servers. This particularly affects highly parallelizable applications, which must achieve efficient cache utilization to maintain performance as CPU core counts grow. Optimizing cache utilization, however, is complex for recent chiplet-based CPUs, whose partitioned L3 caches lead to varying latencies and bandwidths, even within a single NUMA domain. Classical NUMA optimizations and task scheduling fail to address the performance issues of chiplet-based CPUs.

We describe Chiplet Heterogeneity Aware Runtime Mapping (CHARM), a new runtime system designed for chiplet-based CPUs. CHARM combines chiplet-aware task scheduling heuristics, hardware-aware memory allocation, and fine-grained performance monitoring to optimize workload execution. It implements a lightweight concurrency model that combines user-level threading features, such as individual stacks, per-task scheduling, and state management, with coroutine-like behavior, allowing tasks to suspend and resume execution at defined points while efficiently managing task migration across chiplets. Our evaluation across diverse scenarios shows CHARM's effectiveness in optimizing the performance of memory-intensive parallel applications.

ACM Reference Format:
Alessandro Fogli, Bo Zhao, Peter Pietzuch, and Jana Giceva. 2026. CHARM: Chiplet Heterogeneity-Aware Runtime Mapping System. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3767295.3769390

EUROSYS '26, April 27–30, 2026, Edinburgh, Scotland, UK

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769390

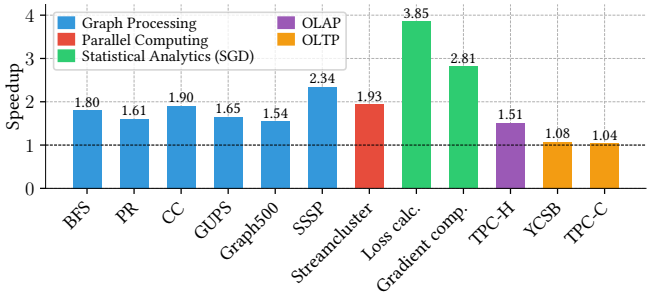


Fig. 1: CHARM speedups compared to NUMA-aware systems across various benchmarks and workloads

1 Introduction

The rapid evolution of server hardware has led to a growing disparity between CPU core counts and the available memory bandwidth. This widening gap between compute power and memory resources particularly affects parallelizable applications, which require efficient memory accesses. As CPU core counts continue to increase, the competition for memory bandwidth intensifies, making cache utilization critical for maintaining application performance [21, 45, 46].

To address these challenges, researchers have explored various approaches, with a significant focus on non-uniform memory access (NUMA) architectures [2, 18, 33]. NUMA systems attempt to bridge the gap between compute power and memory resources by providing localized memory access for each core, thus reducing contention for memory bandwidth and interconnect communication. For example, Lozi et al. [23] analyze the Linux scheduler on NUMA systems and highlight issues with load balancing across CPU cores. Leis et al. [20] introduce morsel-driven parallelism, a NUMA-aware query evaluation framework that dynamically assigns small fragments of input data to worker threads. Kaestle et al. [17] propose Shoal, a system that automatically allocates and replicates data in NUMA systems.

While such NUMA-focused solutions are valuable, the landscape of CPU architectures has evolved significantly

with the introduction of *chiplet-based CPUs* [14, 24, 28, 29, 44]. Chiplet-based CPUs adopt a modular design that consists of multiple smaller dies, or *chiplets*, interconnected to form a larger, more powerful processor. This offers superior flexibility and scalability, allowing manufacturers to create a wide range of processor configurations tailored to performance and power requirements. Consequently, major vendors, such as AMD, Intel, and ARM, adopt this approach in their latest CPU designs [12, 14, 24, 28, 30].

Chiplet-based CPUs, however, also present new challenges in terms of the memory hierarchy and inter-core communication, even within the same NUMA domain [9, 13, 22, 49]. The partitioned nature of the L3 cache, which is distributed across multiple chiplets, introduces a trade-off between cache locality and total cache sizes [9]. Tasks scheduled on fewer chiplets benefit from lower latencies due to local cache accesses but have limited cache capacity. Conversely, spreading tasks across multiple chiplets increases available cache sizes at the cost of higher access latencies. Such heterogeneity extends to inter-core communication, with a wide range of latencies and bandwidths between cores, depending on their location within the chiplet architecture. As a result, a poor task scheduling policy can reduce performance or even result in degrading performance with more cores, and is particularly acute for memory-intensive workloads. Optimizing cache usage and memory allocation for chiplet-based architectures must therefore consider such hardware features and dynamically adapt to changing workloads.

To address these challenges, we describe *Chiplet Heterogeneity Aware Runtime Mapping* (CHARM), a new runtime system designed for chiplet-based CPUs. CHARM uses chiplet-aware task scheduling heuristics, adaptive cache partitioning, coroutine-based task management and fine-grained performance monitoring to optimize workload execution:

(1) Chiplet-aware task scheduling heuristics: CHARM employs heuristics that optimize task placement based on cache affinity and inter-chiplet latencies with a focus on minimizing inter-chiplet communication overhead.

(2) Adaptive cache partitioning: CHARM dynamically adjusts cache allocations based on workload characteristics. This helps balance cache locality and utilization in response to changing workload demands.

(3) Fine-grained parallelism: CHARM utilizes lightweight coroutines that offer a low context switching overhead. This enables efficient management of concurrency, which is particularly beneficial for chiplet architectures that benefit from minimizing the overhead of task switching.

(4) Performance profiling and optimization: CHARM continuously monitors and analyzes application performance metrics. With low-overhead instrumentation, it collects data on computational load and communication patterns, which it uses to make dynamic runtime decisions, such as task migration and scheduling adjustments.

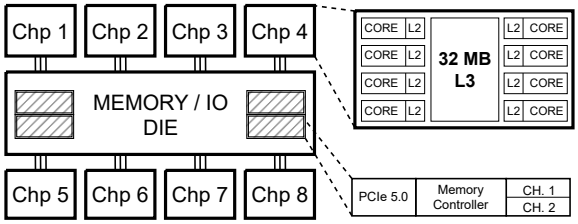


Fig. 2: AMD EPYC Milan

We evaluate CHARM across diverse computational scenarios, including graph processing, statistical analytics, analytical databases, and highly parallel workloads. Based on our evaluation, we provide insights on how to design and configure future systems to best exploit chiplet-based CPUs:

1. Chiplet-aware task partitioning is particularly effective for workloads with irregular memory access patterns, such as graph processing.

2. For OLAP workloads, a hybrid cache partitioning strategy often outperforms strictly local approaches.

3. Higher core counts amplify synchronization and inter-chiplet communication overheads, affecting performance. This is particularly noticeable in statistical analytics for machine learning tasks.

4. Overly strict NUMA-aware optimizations can significantly harm performance on chiplet-based CPUs.

The rest of the paper is structured as follows:

- We give background on chiplet-based CPU architectures and analyze their inter-core latencies (§2).
- We present the challenges of implementing effective chiplet-aware scheduling strategies (§3).
- We describe the architecture of CHARM, including its core components, chiplet-aware task scheduling heuristics, and monitoring strategies (§4).
- We present a comprehensive evaluation of CHARM with various benchmarks and applications (§5).

2 Chiplet-Based CPUs

Chiplet-based CPUs are a significant advancement in processor design: they utilize modular systems with interconnected smaller dies, or *chiplets*, to create more powerful processors. This approach offers benefits over traditional monolithic designs, including enhanced scalability, better manufacturing yields, and reduced costs [9]. The flexibility of chiplet designs allows manufacturers to combine various chiplets, meeting different performance and power requirements, and simplifying upgrades by enabling individual chiplet replacement or updates [49]. For these reasons, in recent years, major companies such as AMD, Intel, and ARM adopted this technology in their latest processors [12, 24, 30].

Fig. 2 shows the architecture of a single-socket AMD EPYC Milan CPU. It uses multiple core complex dies (CCDs), i.e., chiplets, connected to a central I/O die via AMD's Infinity

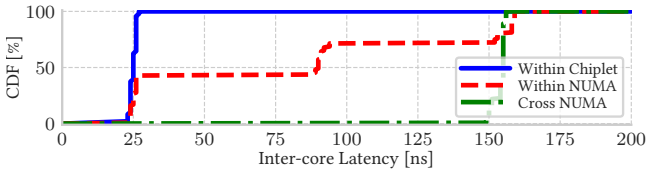


Fig. 3: Cumulative distribution function (CDF) of core-to-core latency in AMD EPYC Milan CPUs

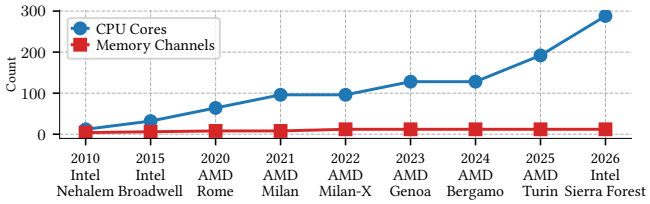


Fig. 4: Number of memory channels vs. cores over the years

Fabric interconnect. Each CCD contains one or two core complexes (CCXs), depending on the processor generation. Each CCX consists of multiple cores, each paired with its own L2 cache, surrounded by a shared 32 MB L3 cache. The aggregate L3 cache is distributed across the chiplets, rather than being a single unified cache for the entire processor.

2.1 Inter-core latencies

The performance of chiplet-based CPUs is affected significantly by non-uniform access times to the L3 cache and by the varying latencies and bandwidths between cores. We measure core-to-core latencies on a dual-socket AMD EPYC Milan processor, assigning threads to different cores and using compare-and-swap (CAS) operations to measure communication delays. The cumulative distribution function (CDF) in Fig. 3 shows the results. We observe a clear latency hierarchy: communication between cores on the same chiplet is fastest, followed by communication within the same NUMA node. The slowest latencies occur across NUMA nodes, reflecting the fact that on-chip communication is significantly faster than inter-chip communication.

In contrast to typical assumptions, the "within NUMA" curve shows greater variability. There are three clear groupings of latencies: the lowest around 25 ns represents intra-chiplet communication; a middle group around 80–90 ns indicates inter-chiplet but intra-CCX (core complex) communication; and a higher group beyond 150 ns represents communication across different CCXs within the same NUMA node. This stepped distribution highlights the heterogeneous nature of core-to-core latencies within a single NUMA domain in chiplet-based CPUs and can have a significant impact on performance. This complicates task allocation and resource assignment and can lead to imbalances in utilization.

2.2 More cores, limited memory channels

Using chiplets, manufacturers can pack more cores into a single package, enhancing computational power and efficiency.

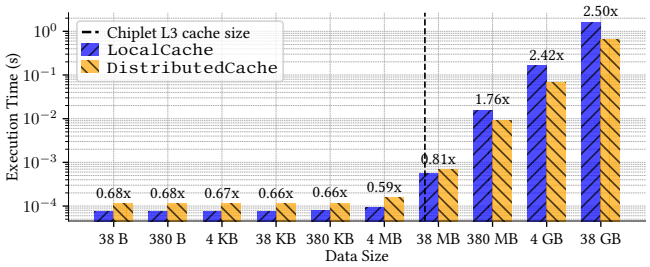


Fig. 5: `LocalCache` vs. `DistributedCache`: write operation speedup when varying the data array size (with 8 chiplets)

This strategy, however, does not help with the limited memory bandwidth per core. Fig. 4 shows this increasing disparity: while in 2010, high-end server processors typically had 4–8 cores, as of 2023, AMD's EPYC Genoa processors offer up to 96 cores. In contrast, memory channel growth has not kept pace: current high-end CPUs have up to 8–12 memory channels. This trend is expected to continue: by 2026, we may see CPUs with 300 cores but not more memory channels.

The increase in memory bandwidth per channel has also not kept pace with the rise in core counts. For example, DDR5 roughly doubles the bandwidth of DDR4, but core counts have increased by an order of magnitude since 2010. As a result, the available bandwidth per core continues to decline, even as the total socket bandwidth improves.

Looking at the cache hierarchy, we see that chiplet designs have enabled significant increases in L3 cache capacity (e.g., AMD's EPYC processors have up to 256 MB of L3). This helps mitigate some of the memory bandwidth limitations but comes with an increased heterogeneity of latencies. This requires us to adjust how we run applications to unlock the full potential of chiplet-based CPUs.

2.3 Parallel processing on chiplets

The increasing core counts and limited memory bandwidth present challenges for parallel processing on chiplet-based CPUs. As more cores compete for the same memory resources, intelligent utilization of on-chip caches becomes crucial to maintain high performance. This scenario raises important questions about how to best leverage the distributed nature of L3 caches across multiple chiplets.

We investigate the performance of two caching strategies on multi-chiplet CPUs: `LocalCache` and `Distributed-Cache`. In our setup, `LocalCache` confines data access to CPU cores within a one chiplet, thereby leveraging its local L3 cache; `DistributedCache` distributes the same number of cores across multiple chiplets, utilizing the collective L3 cache capacity but incurring inter-chiplet communication overhead.

To understand these effects, we conduct a microbenchmark on a single-socket 8-chiplet AMD EPYC Milan CPU. We measure the execution time of a multithreaded (8 threads) write to a vector, divided into contiguous equal-sized segments. Each thread is assigned to a dedicated core and calculates the start and end indices of its assigned segment. We

report the mean of 1,000 iterations, varying the data size from 38 B to 38 GB. As a warm-up, each thread performs a pass over its segment, setting all elements to 1.

Fig. 5 shows the results on a logarithmic scale: for data sizes up to 38 MB, `LocalCache` has superior performance compared to `DistributedCache` with lower execution times due to the avoidance of inter-chiplet communication. This advantage diminishes beyond 32 MB due to limited L3 cache capacity. At that point, `DistributedCache` becomes more effective, peaking at a 2.5× speedup for the 38 GB dataset. The performance difference between the two strategies ranges from 0.6× to 2.5×, and the optimal choice depends on the workload's working set size and access patterns.

3 Challenges With Chiplets

Runtime systems are an effective solution to the challenges of parallel computing in modern hardware environments. At their core, these systems decompose applications into discrete units of work, or tasks, with well-defined data inputs and outputs. Rather than directly calling computation kernels, developers define tasks and their dependencies, allowing the runtime system to distribute work dynamically across available compute resources.

The strength of task-based systems lies in their dynamic scheduling approach: they can integrate detailed information about task memory footprints, dependencies, and execution characteristics. This allows for more informed scheduling decisions, reducing scheduling overheads and improving the management of memory hierarchies. Such optimizations are especially relevant for highly parallel tasks, for which inefficient scheduling can lead to performance degradation due to load imbalances and NUMA effects.

A chiplet-aware runtime must address several challenges:

(1) Cache management: Unlike memory, users cannot directly allocate data in chiplet caches. Hence, traditional runtimes that have focused primarily on memory allocation in NUMA nodes or thread placement based on core availability and workload distribution need to be extended to consider the trade-off between cache locality and cache availability when the cache is partitioned (see §2).

(2) Inter-chiplet communication and synchronization: The increase in latency between chiplets (described in §2.1) requires optimized task mapping and scheduling. Runtime systems must minimize data movement across chiplets, but also implement efficient and lightweight synchronization mechanisms to prevent bottlenecks. Traditional thread-based methods can suffer due to overheads from OS thread creation, context switching, and management. This can be problematic for fine-grained tasks as the number of chiplets grows.

(3) Workload adaptation: A single policy cannot address the diverse needs of all applications: some are latency-bound, others are memory-intensive, and their working set sizes can vary during execution. This requires adaptive resource management (see §2.3). For example, a workload may initially benefit from high data locality, keeping related data close. However, as the working set expands, the same workload may require increased cache availability to maintain performance. Effective chiplet-aware runtime systems must recognize these shifting demands and adjust the resource allocation accordingly throughout execution.

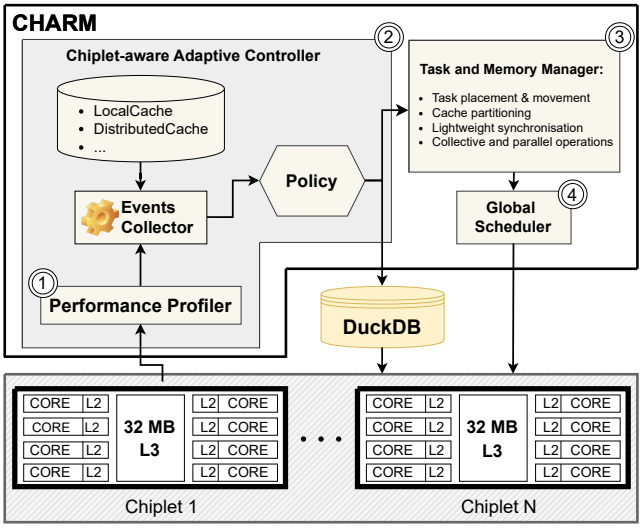


Fig. 6: CHARM architecture

4 CHARM System Design

We describe the design and implementation of CHARM, a runtime system for modern chiplet-based CPUs. At its core, CHARM revolves around two fundamental pillars: (1) **lightweight task management**: CHARM implements nimble mechanisms for fine-grained task execution across chiplets (see §4.2 and §4.3), seamlessly integrating with contemporary runtime systems; (2) **intelligent placement decisions**: CHARM makes informed choices about task movement (see §4.4) by incorporating chiplet-aware objectives.

To handle the variability of workloads, CHARM adopts an adaptive approach. It incorporates a lightweight profiling component that gathers key metrics about task behavior, including memory access patterns and communication frequency. It then dynamically adjusts task placement at runtime to adapt to workload changes and system conditions.

4.1 CHARM architecture

The CHARM system is designed with four key components. As shown in Fig. 6, the architecture includes: (1) a performance profiler; (2) an adaptive controller; (3) a memory and task manager; and (4) a global scheduler.

The **performance profiler** (1) continuously monitors and analyzes application performance metrics. Using low-overhead instrumentation, CHARM collects detailed data on computational load and communication patterns. During workload execution, it periodically checks the frequency

of accesses to the local chiplet, remote chiplets, and main memory. This helps determine the cache availability. The profiler can also monitor only specific code segments, providing detailed and accurate results for individual tasks or threads with minimal overhead.

The **adaptive controller** ② gathers information from the profiler and uses predefined approaches to generate scheduling policies. An *approach* outlines the general method or guiding principle, while a *policy* specifies the concrete actions the scheduler follows based on that approach. For example, a location-centric approach may focus on minimizing cross-chiplet communication, while the corresponding policy dictates the assignment of tasks to cores within the same chiplet. These predefined approaches can be extended to create more precise policies tailored to workload requirements. In our prototype system, the controller generates adaptive policies that switch between location- and cache size-centric approaches. These policies dynamically balance the benefits of local cache accesses with the need for larger aggregate cache sizes. This adaptive partitioning approach allows CHARM to respond to evolving workload demands, ensuring high performance and efficiency.

The **task and memory manager** ③ handles tasks from the user application using lightweight coroutines. Coroutines are allocated or shifted to specific CPU cores, enabling the management of numerous concurrent tasks and enhancing parallel execution and overall application throughput. The system supports NUMA-aware allocation to optimize main memory access patterns. Barrier synchronization mechanisms also coordinate task execution across multiple chiplets.

Finally, the **global scheduler** ④ coordinates task distribution and load balancing across available compute resources. It receives policies from the adaptive controller and interfaces with the task and memory manager to obtain lightweight coroutines representing individual tasks. It also maintains a global system view and can quickly respond to changes in workload patterns by receiving new policies and migrating tasks between chiplets as needed, which ensures that the system remains optimized under varying conditions. The global scheduler is not preemptive. Tasks can yield execution voluntarily at well-defined points, such as when waiting for data. The scheduler strives to preserve the initial task-to-worker-to-core mapping as much as possible.

CHARM differs from centralized scheduling paradigms by decentralizing the decision-making process: each worker thread independently monitors its own performance metrics. Based on these local observations (e.g., the rate of remote cache accesses), each worker decides autonomously whether to adjust its CPU affinity to spread across more chiplets or to concentrate on fewer. It then informs the task manager and global scheduler to enact this change. This approach eliminates global data collection and centralized decision-making, thereby reducing overheads and improving responsiveness.

Algorithm 1 Chiplet Scheduling Policy

```
1: procedure ChipletScheduling
2:     current_time ← steady_clock.now()
3:     elapsed ← current_time − time
4:     if elapsed ≥ SCHEDULER_TIMER then
5:         counter ← getEventCounter()                ▷ Cache fill events
6:         rate ← counter × SCHEDULER_TIMER/elapsed
7:         if rate ≥ RMT_CHIP_ACCESS_RATE then
8:             if spread_rate < CHIPLETS then
9:                 spread_rate ← spread_rate + 1
10:            end if
11:        else
12:            if spread_rate > 1 then
13:                spread_rate ← spread_rate − 1
14:            end if
15:        end if
16:        updateLocation()              ▷ Spread or compact on chiplets
17:        time ← steady_clock.now()
18:        resetEventCounter()
19:    end if
20: end procedure
```

4.2 Chiplet-aware task scheduling policy

Maximizing cache utilization is important for achieving high performance on chiplet-based systems with complex memory hierarchies. To this end, CHARM incorporates heuristics to improve cache affinity when scheduling and migrating tasks. These heuristics aim to keep related data and computations co-located to minimize expensive data movements. To handle more dynamic workloads, CHARM also implements an adaptive cache partitioning scheme. This scheme monitors the cache usage of different task types and adjusts the allocation of cache resources to maximize overall system throughput. The partitioning is periodically refined based on runtime performance measurements.

The `Chiplet Scheduling Policy` (Alg. 1) governs this adaptive behavior. The algorithm operates periodically on a per-worker basis. It begins by checking the time elapsed since the last decision. If enough time has passed, the worker retrieves its local cache fill event counter, which tracks remote memory accesses. It then calculates the rate of these accesses and compares it to a predefined threshold. This threshold can vary depending on the approach used, e.g., a higher value would delay changes to the scheduling.

CHARM implements a decentralized scheduling paradigm in which each worker thread independently decides on migration based on local observations. Specifically, each worker employs a local parameter, `spread_rate`, to configure the number of chiplets to distribute its tasks to. `spread_rate` is initialized to one. When the remote access rate exceeds the threshold, the worker increments its `spread_rate` to expand the footprint across more chiplets for larger dedicated cache capacity. Conversely, if the rate is low, the worker decrements the value to consolidate its tasks onto fewer chiplets, thus improving data locality with other threads.

Algorithm 2 Update Task Location

```
1: procedure UpdateLocation
2:     if spread_rate ∉ (0, CHIPLETS] or THREAD_SIZE ≥ spread_rate × CORES_PER_CHIPLET then
3:         return                                                    ▷ Bounds check
4:     end if
5:     chiplet ← ⌊ unique_worker_ID / (CORES_PER_CHIPLET/spread_rate) ⌋
6:     slot ← unique_worker_ID mod (CORES_PER_CHIPLET / spread_rate)
7:     if chiplet ≥ CHIPLETS then
8:         chiplet ← chiplet mod CHIPLETS
9:         slot ← slot + ⌊ unique_worker_ID / CORES_PER_CHIPLET ⌋
10:    end if
11:    core ← chiplet × CORES_PER_CHIPLET + slot
12:    set_thread_affinity(core)                                     ▷ Set affinity
13:    numa_node ← ⌊ core / CORES_PER_NUMA_NODE ⌋
14:    set_mempolicy(MPOL_BIND, 1 ≪ numa_node)                       ▷ Set memory policy
15: end procedure
```

Such a decentralized approach prevents resource contention without a central arbiter. After updating a worker's `spread_rate`, the algorithm invokes `UpdateLocation()` to calculate a unique deterministic core assignment. This ensures a final collision-free core mapping, as described below.

4.3 Adaptive cache partitioning

The value of `spread_rate` is determined by the following trade-off; on the one hand, consolidating tasks by reducing the spread enhances data locality, which is ideal for workloads with high data sharing; on the other hand, spreading tasks by increasing the rate provides a larger aggregate cache, which is better suited for workloads with large, independent data sets. The `UpdateLocation` algorithm (Alg. 2) is responsible for making this decision, translating the high-level policy described in §4.2 into a collision-free task placement.

The algorithm first performs a sanity check to prevent invalid remappings. It ensures that the `spread_rate` is within the valid range of physical chiplets and the resulting configuration provides sufficient unique physical cores to accommodate all worker threads. This check is important, because CHARM dedicates one physical core to each worker thread to prevent resource contention. For example, given 64 worker threads and a chiplet with 8 cores, a `spread_rate` of 1 is invalid, because it is impossible to map 64 threads to 8 dedicated cores on a single chiplet. If this check fails, the worker's migration is skipped, its core affinity remains unchanged, and it simply retries the evaluation in the next timer cycle.

If the check passes, the algorithm calculates the target chiplet and core slots for each worker. Since each worker is assigned a unique ID, the resulting (chiplet, slot) pair is also unique, guaranteeing a deterministic and collision-free core assignment. The algorithm then sets the thread's affinity and binds its memory allocation to the corresponding NUMA node, taking precedence over the OS's NUMA balancing to minimize remote memory accesses. Note that, to minimize the impact of data migration, CHARM performs such updates upon task completion and only when significant inefficiency is detected. The combination of these two algorithms allows CHARM to adjust its task distribution strategy dynamically based on observed cache usage patterns.

4.4 Fine-grained task parallelism

Traditional OS-managed threads can introduce high overheads during context switching and synchronization, which is particularly severe when managing a high volume of tasks across multiple chiplets.

To overcome these challenges, CHARM implements a lightweight concurrency system that combines features of user-level threads and coroutines. Analogous to user-level threads, CHARM tasks have their own scheduler, individual stacks for each execution unit, and state management. This design allows CHARM to handle task movement across chiplets efficiently, managing execution contexts independently of the OS. At the same time, CHARM incorporates coroutine-like behavior, particularly the ability to suspend and resume execution at developer-defined points, similar to the concurrency model used in the RING runtime system [25]. For example, when a coroutine yields, CHARM's integrated profiling system activates and analyzes task behavior, memory access patterns, and inter-chiplet communication. The profiling data enables CHARM to adjust task placement across chiplets dynamically, thus optimizing cache locality and minimizing remote memory accesses in real-time.

Within each core, CHARM maintains a local task queue designed for low-overhead operation. Using lock-free mechanisms based on atomic operations, tasks are enqueued and dequeued efficiently by multiple worker threads, avoiding costly synchronization delays. When a coroutine yields or completes, the worker thread checks its local queue for pending tasks. If the worker's local task queue is empty, it uses a work-stealing approach by attempting first to steal tasks from cores on the same chiplet before considering other chiplets. This strategy helps preserve cache locality, which is critical for efficient chiplet-based systems.

4.5 Performance profiling and optimization

Monitoring in CHARM provides real-time insights into the system performance, enabling dynamic adjustments to the scheduling and memory management policies. It collects performance data in user space, which minimizes context switches and overheads. CHARM provides a set of performance counters that can be used to monitor various metrics, such as cache misses, memory bandwidth utilization, and task execution times. To achieve this, CHARM uses *libpfm* [7] and relies on existing performance monitoring unit (PMU) counters to collect profiling data, which avoids

manual code instrumentation and ensures minimal overhead. This overhead, which can be around 5–10%, corresponds to high-frequency polling; users, however, can customize this frequency to reduce impact, making it suitable for a wider range of use cases in which a different balance between insights and performance is needed. More specifically, CHARM tracks chiplet cache fill rates using `ANY_DATA_CACHE_FILLS_FROM_SYSTEM` on AMD systems and `OFFCORE_RESPONSE` on Intel systems to distinguish fills from on-chip (intra-CCX), on-die (inter-CCX), and remote memory sources (inter-NUMA). On Intel systems, we configure event filter masks that target last level cache (LLC) hits and DRAM responses from local and remote sources. These counters, while limited in granularity, enable us to differentiate communication across chiplets and NUMA boundaries.

4.6 Implementation and API

CHARM is implemented as a C++ software framework that enables efficient mapping and execution of applications on chiplet-based CPUs. Based on the work of Grappa [31] and RING [25], the design is highly modular, with separate components handling different aspects of the system, such as task scheduling, memory management, communication, and profiling. Each component can be reused, reconfigured, or replaced independently, which allows for a high degree of customization and optimization tailored to specific application requirements. Specifically, CHARM adds a real-time profiler, a scheduler aware of chiplet constraints, coroutine support, and a work-stealing mechanism. We have kept RING's original API and task/RPC model.

CHARM targets single-application scenarios in which the source code is available and assumes exclusive hardware access to maximize resource utilization and avoid contention. Although this simplifies scheduling, recent work suggests that chiplet-aware strategies can also benefit multi-tenant, shared-nothing environments [9]. CHARM assumes a symmetric chiplet layout with uniform memory access and does not model chiplet-to-memory-controller distances, because many architectures integrate memory controllers within chiplets to mitigate such effects.

CHARM provides developers with a straightforward and efficient API for parallel programming on chiplet-based architectures. CHARM is initialized using `CHARM_Init()` and cleaned up with `CHARM_Finalize()`. Developers can define parallel tasks using lambda functions within the `run()` function. CHARM offers various task execution methods, including `all_do()` for executing a task on all cores, and `call()` for remote procedure calls with both synchronous and asynchronous options. When executing these tasks, developers can focus on application logic while CHARM handles the complexities of chiplet-aware resource allocation, task distribution, and performance optimization. CHARM also provides synchronization primitives, such as `barrier()`, to ensure that parallel tasks are coordinated effectively.

CHARM avoids assigning concurrent tasks to hyperthreads on the same physical core to prevent contention in the L1 and L2 caches. To maintain cache isolation, it treats each physical core as the smallest independent scheduling unit. In addition, for multi-level NUMA, CHARM applies a socket-aware policy: it fully utilizes all cores and chiplets within one socket before utilizing another. This preserves cache locality and keeps memory accesses within the same NUMA domain, minimizing costly inter-socket communication.

Before reporting our evaluation of CHARM, we perform a sensitivity analysis that systematically varies the candidate threshold value and measures its impact on key performance metrics, including throughput, latency, and cache utilization. Based on this analysis, we determine that an `RMT_CHIP_ACCESS_RATE` of 300 events per `SCHEDULER_TIMER` interval provides the best balance of performance across our experimental scenarios. This threshold effectively triggers task redistribution when inter-chiplet communication becomes excessive, while also allowing sufficient task consolidation to benefit from cache locality.

All experiments use a `SCHEDULER_TIMER` interval of 500 ms, which is chosen to balance profiling overhead and responsiveness. This interval is short enough to capture access pattern changes without interfering with execution. The scheduler runs periodically and preserves initial task-to-core mappings unless profiling reveals issues, such as remote chiplet access or cache contention. When adaptation is needed, CHARM rebalances tasks incrementally in a topology-aware manner, minimizing data movement and preserving affinity. It follows a hierarchical approach: expanding execution outwards to nearby chiplets as working sets grow and contracting inwards when locality can be regained.

5 Evaluation

We conduct experiments to evaluate CHARM's efficacy on modern chiplet-based architectures over a range of benchmarks and applications. In particular, we address the following questions:

- **Q1:** How does CHARM's performance compare to existing runtime systems, such as RING [25], and NUMA-aware schedulers, such as AsymSched and SAM, across a range of computational tasks? (§5.2)
- **Q2:** Does CHARM keep the performance advantage across different processor architectures (e.g., AMD/Intel)? (§5.3)
- **Q3:** What are the performance trade-offs and scalability characteristics of CHARM compared to optimized NUMA-aware systems on chiplet-based architectures? (§5.4)
- **Q4:** Does CHARM exploit the heterogeneity of chiplet-based architectures to accelerate irregular workloads, such as graph processing and sparse linear algebra? (§5.5)
- **Q5:** How does CHARM accelerate data-intensive analytical workloads? (§5.6)

5.1 Experimental setup

Testbed. We have conducted experiments on two platforms:

1. a dual-socket AMD EPYC Milan 7713 processor. Each socket features 64 CPU cores, 512 GB RAM and 8 chiplets; each chiplet is equipped with a 32 MB local L3 cache. The codebase is compiled with GCC 12 with the `-O3` optimization flag, running on Ubuntu Linux 23.04.
2. a dual-socket Intel Xeon Platinum 8488C processor, with each socket housing 48 cores (96 threads), 512 GB of RAM, and a shared 105 MB L3 cache. The system operates on Ubuntu Linux 22.04.5 LTS, and the codebase was compiled using GCC 12 with the `-O3` optimization flag.

While ARM-based processors (e.g., the Neoverse N1) become more common on cloud platforms, their architecture makes them less relevant to this work. The Neoverse N1 design concentrates all compute cores on a single chiplet, giving a monolithic layout. Since prior work has shown that such designs have minimal sensitivity to chiplet-aware scheduling strategies [9], we focus our evaluation on platforms for which these optimizations are most relevant.

Baselines. We compare the performance of CHARM against four systems:

1. RING [25] is a NUMA-aware, message-batching runtime system designed for high-performance and in-memory data-intensive workloads.
2. SHOAL [17] is a runtime system that provides an array abstraction for optimized memory allocation and access patterns on NUMA multi-core architectures.
3. AsymSched [41] is a bandwidth-centric NUMA scheduler designed to optimize thread and memory placement with asymmetric interconnects.
4. SAM [38] is a multicore CPU scheduler for modern multiprogrammed machines that mitigates performance issues from data sharing and contention by identifying latency tolerances and incorporating hyperthreading awareness.

Benchmarks. To evaluate CHARM and the baseline systems, we use workloads from four domains: (i) graph processing; (ii) high performance parallel processing; (iii) statistical analytics; and (iv) database management. Specifically, we use the following benchmarks and applications[1]:

- *RandomAccess* evaluates non-contiguous memory accesses in a distributed shared memory architecture, measured in global updates per second (GUPS).
- *Graph algorithms* have irregular access patterns. We use five graph algorithms: Breadth-First Search (BFS) [26], Connected Component (CC) [48], Single Source Shortest Path (SSSP) [40], PageRank (PR) [1], and Graph500 [27]. The graph is a Kronecker graph model with 2^{24} vertices and 16×2^{24} edges, of approximately 4 GB.
- *Statistical analytics* uses stochastic gradient descent (SGD) on a dataset with 10,000 samples and 8,192 features, totaling approximately 6,250 MB. For this evaluation, we use DimmWitted, an analytics engine optimized for statistical computations on modern NUMA architectures [51].
- *OLAP* evaluates TPC-H [43], with a scale factor of 100, using the DuckDB [35] engine with and without the CHARM adaptive controller module.
- *OLTP* evaluates YCSB [6] and TPC-C [42] using the ERMIA engine [19]. YCSB is configured with 50 million records in a single table, running a mixed workload of 45% read and 55% read-modify-write operations. TPC-C simulates 50 warehouses with a workload of 45% New Order, 43% Payment, and smaller proportions of Delivery, Order Status, and Stock Level transactions. It supports cross-partition transactions, uses a uniform item distribution, and always accesses the home warehouse. The implementation includes 10,000 suppliers without specialized scan optimizations for Order Status.
- *Streamcluster* [50] runs compute-intensive clustering algorithms sensitive to memory access patterns. We use it to compare CHARM with SHOAL[17] in parallel processing scenarios on shared-memory multicore architectures. Streamcluster provides insights into working sets, locality, data sharing, synchronization, and off-chip traffic. Experiments process 1 million data points with 128 dimensions, targeting 10–20 cluster centers, and allowing up to 5,000 intermediate centers, with data chunked into 200,000-point batches.

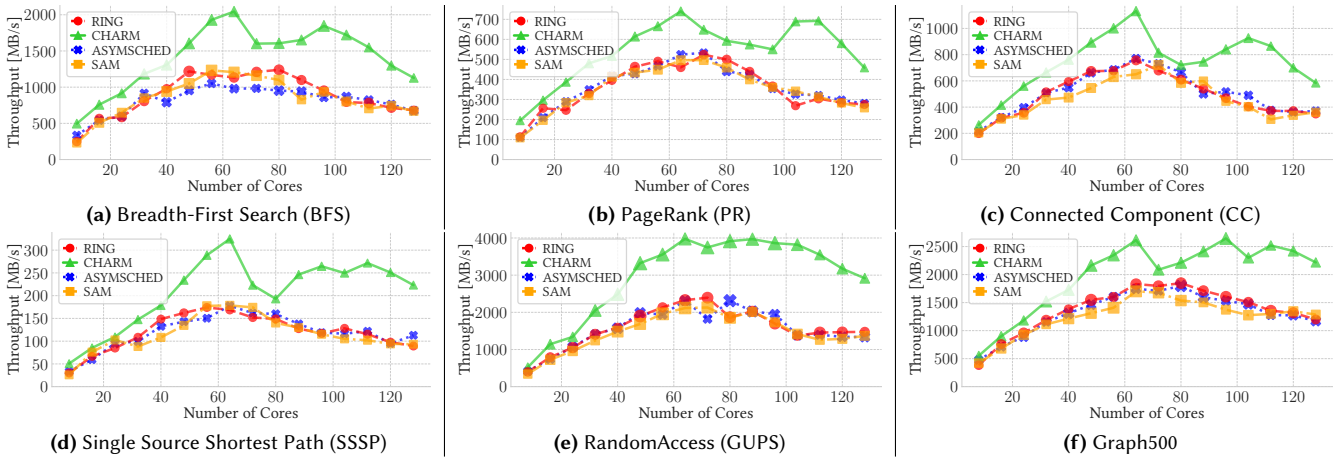
Integration. Integrating CHARM into target applications is straightforward and requires only minimal code refactoring. It involves: (i) linking the CHARM library; (ii) initializing its components at startup; and (iii) wrapping existing task logic with CHARM's API.

For example, the integration with DimmWitted only requires identifying task boundaries with the `Charm::call` interface, without significant refactoring. In contrast, DuckDB's more complex architecture requires modifying the scheduler to leverage CHARM's profiler and adaptivity.

Measurements. We measure three performance metrics: (i) throughput; (ii) speedup of CHARM over baseline systems; and (iii) memory bandwidth utilization. The results are averages over 10 runs. All experiments are conducted on the AMD-based machine unless stated otherwise.

We also decompose tasks for all benchmarks: graph analytics (e.g., BFS, PageRank, Connected Components) generates tasks dynamically per active frontier node; DimmWitted partitions workloads into hundreds of fine-grained chunks across over 600 threads. Execution times vary by workload and input size. We evaluate CHARM across diverse dataset sizes and runtimes. Graph benchmarks run until full traversal or convergence, ranging from seconds to minutes depending on the graph scale.

[1]For reproducibility, the source code and all artifacts presented in this paper are available at https://github.com/Alessandro727/CHARM.



(a) Breadth-First Search (BFS) **(b)** PageRank (PR) **(c)** Connected Component (CC)

(d) Single Source Shortest Path (SSSP) **(e)** RandomAccess (GUPS) **(f)** Graph500

Fig. 7: Graph Processing + Random Access Scalability on AMD machine (Marker sizes are scaled to represent the variance.)

5.2 Overall effectiveness and efficiency

We first investigate the overall effectiveness and efficiency of CHARM. We examine the scalability of the processing throughput of six algorithms: Breadth-First Search (BFS); PageRank (PR); Connected Component (CC); Single Source Shortest Path (SSSP); RandomAccess (GUPS); and Graph500. We compare CHARM's scalability against the baselines of RING, AsymSched, and SAM.

Fig. 7 shows the results. We observe that CHARM achieves near-linear scaling, up to 64 cores, consistently outpacing all baselines. Beyond 64 cores, its growth tapers slightly as our scheduler fills all chiplets within one socket before assigning tasks to the other, temporarily unbalancing core assignments across sockets. Bandwidth utilization, however, rebounds between 96 and 104 cores once the imbalance decreases. In contrast, RING, AsymSched, and SAM all saturate around 48–56 cores, reflecting their limitation in fully exploiting intra-chiplet memory bandwidth. Since these systems all rely on NUMA-aware policies, they offer similar performance and lack the fine-grained chiplet locality of CHARM.

With 64 cores, RING, AsymSched and SAM peak, and the most pronounced improvements appear in BFS, CC, and SSSP: CHARM delivers 1.8×, 1.9×, and 2.3× speedups over the baselines, respectively. With 96 cores and above, CHARM's advantage grows further—to between 2× and 2.8×—as the baselines' throughput declines under contention.

We attribute CHARM's performance gain to the advanced chiplet-aware task scheduling policy to optimize inter-chiplet communication and the load balancing strategies that exploit the local L3 cache. While NUMA-aware schedulers avoid remote memory allocation and accesses, they fail to prevent L3 cache access from remote NUMA domains. This results in increased communication overhead (i.e., data movement) when accessing L3 caches across multiple chiplets and sockets.

The effect becomes more pronounced at higher core counts across all benchmarks (see Fig. 7). CHARM reduces remote NUMA chiplet accesses (see Tab. 1) by collocating tasks and data within local chiplets, avoiding the NUMA-negative effect. With 64 cores per NUMA node, CHARM fully occupies each socket, eliminating "free spots" and minimizing cross-chiplet task movement. In the SSSP benchmark, CHARM records 1.5×10^8 local accesses versus RING's 8.7×10^4, and only 6×10^3 remote accesses compared to RING's 2.3×10^8. This balance between task placement and cache locality results in CHARM's superior scalability.

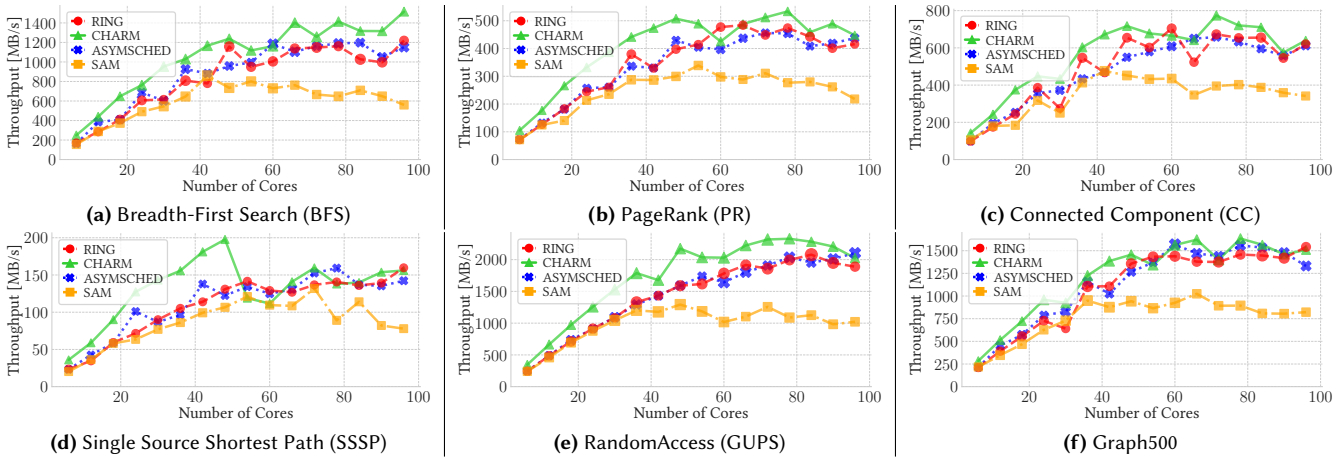
Tab. 1: Comparison of chiplet accesses ($\times 10^3$) using 64 cores

Application	Remote NUMA Chiplet		Local Chiplet	
	CHARM	RING	CHARM	RING
BFS	3	20876	24722	14687
PR	9	55960	78561	39212
CC	84	43718	54631	27924
SSSP	6	230939	153665	87152
GUPS	25	19377	21033	18328
Graph500	93	93196	229608	176853

In summary, CHARM efficiently leverages chiplet-based architectures through optimized task scheduling and load balancing. Its linear scalability in iterative algorithms (e.g., PageRank) reflects effective synchronization that minimizes global update overhead and enhances cross-chiplet parallelism. Similar trends across diverse graph algorithms demonstrate CHARM's adaptability to varying computational patterns and data dependencies.

5.3 Efficiency across chiplet-based architectures

Next, we investigate whether CHARM maintains its performance advantage across chiplet-based architectures, namely a dual-socket Intel Xeon Platinum 8488C system. Using the full suite of graph-processing benchmarks (BFS, PageRank, Connected Components, SSSP, GUPS, and Graph500) and a



(a) Breadth-First Search (BFS) **(b)** PageRank (PR) **(c)** Connected Component (CC)

(d) Single Source Shortest Path (SSSP) **(e)** RandomAccess (GUPS) **(f)** Graph500

Fig. 8: Graph Processing + Random Access Scalability on Intel machine (Marker sizes are scaled to represent the variance.)

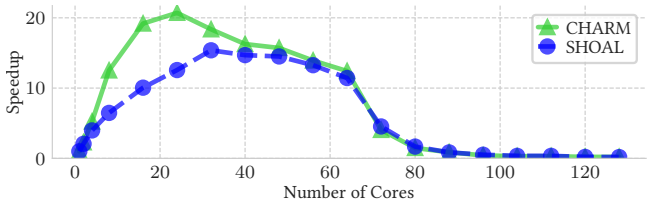


Fig. 9: Scalability analysis of Streamcluster performance: Shoal compared to CHARM

Tab. 2: Comparison of memory and cache accesses ($\times 10^3$) between CHARM and SHOAL across core counts

Cores	Local Chiplet		Local NUMA Remote Chiplet		Main Memory	
	CHARM	SHOAL	CHARM	SHOAL	CHARM	SHOAL
8	27055	11014	718	9	7037	49222
16	30956	27302	893	3623	6941	41273
32	35099	50054	1179	2528	6634	24832
64	48798	67560	1852	1142	5256	5092

random-access workload, we report throughput and scalability in Fig. 8. Here, CHARM outperforms ASYMSCHED, RING, and SAM across all benchmarks, by a large margin up to 48 cores—the capacity of a single socket—when its chiplet-aware scheduling proves most effective.

Beyond 48 cores, as execution spans sockets, CHARM's throughput initially drops then recovers, mirroring the trend on AMD. The performance gap with ASYMSCHED and RING narrows in this range, with CHARM maintaining a slight edge or parity. This reduced advantage is due to architectural differences: AMD employs more chiplets and a higher-bandwidth interconnect than Intel. SAM consistently underperforms on Intel, as its PMU-based heuristics (e.g., IPC and coherence activity) are poorly suited to the architectural intricacies and chiplet-based design features of this platform.

The results across different chiplet-based architectures suggest that, although Intel Sapphire Rapids offers better inter-core communication than AMD EPYC Milan, achieving peak performance still demands a scheduling strategy that takes the processor's topology into account.

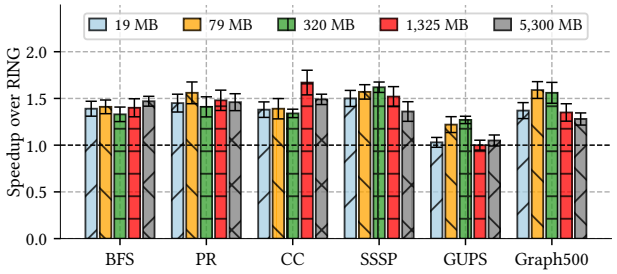
5.4 Scalability with optimized NUMA-aware systems

To evaluate the effect of chiplet-aware scheduling on parallel workloads, we run CHARM on the Streamcluster benchmark [50]. We compare to SHOAL [17], which optimizes memory allocation and replication for NUMA systems using architectural hints (e.g., huge pages and DMA copy engines).

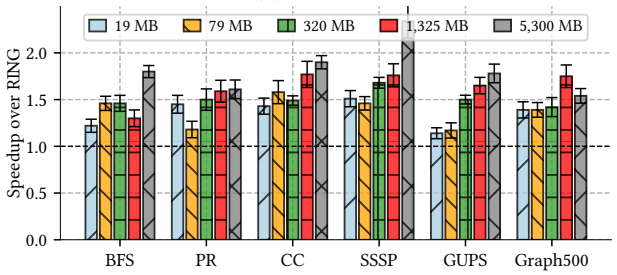
Fig. 9 shows speedups over the execution time of the Streamcluster workload without any architecture-aware runtime support, across 1 to 128 cores. CHARM achieves a peak speedup of 21× with 24 cores, while SHOAL reaches 16× with 32. The largest gap appears with 16 cores, where CHARM outperforms SHOAL by 2×, and CHARM maintains its lead until 40 cores. Beyond that, CHARM and SHOAL's speedups gradually drop to 1× due to high parallelism that fragments the input across too many worker threads. This shrinks each thread's workload to the point at which scheduling optimizations yield negligible benefits.

These trends highlight key architectural differences. SHOAL assigns tasks sequentially to cores (e.g., task 0 to core 0), which works well for NUMA but underutilizes chiplet-based designs such as AMD EPYC Milan. With 16 cores, SHOAL only uses 2 of 8 chiplets—accessing just 64 MB of L3 cache instead of the full 256 MB—limiting performance. In this experiment, we use 1 million points, each with 128 dimensions; each dimension is a 4-byte floating-point number, resulting in a total dataset size of approximately 512 MB, which far exceeds the cache capacity of two chiplets.

Tab. 2 corroborates this: with 8 cores, SHOAL incurs over 7× more main memory accesses than CHARM. With 16 cores, its inter-chiplet traffic increases due to frequent task and data movement across two chiplets. CHARM, by contrast, maintains balanced, locality-aware scheduling with most accesses served from local caches. With 32 cores, SHOAL's remote chiplet accesses surge, while CHARM preserves efficient data locality. With 64 cores, the memory access patterns of both systems converge, indicating similar levels of hardware utilization at high parallelism.

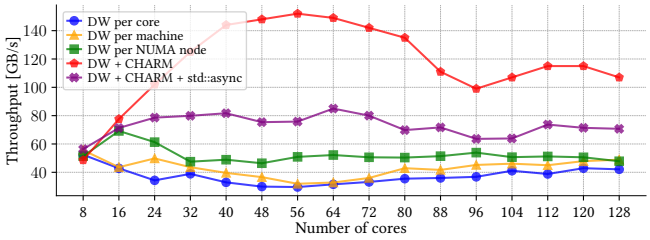


(a) 32 Cores

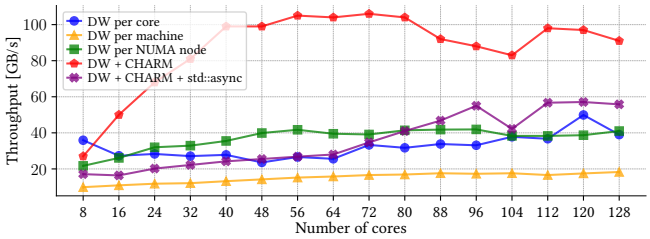


(b) 64 Cores

Fig. 10: CHARM's speedups over RING, varying graph sizes



(a) Logistic loss calculation



(b) Gradient computation and model update

Fig. 11: SGD for logistic regression model

5.5 Sensitivity analysis

In this section, we assess the sensitivity of CHARM's performance with respect to: (i) the data sizes of workloads and (ii) the irregular access patterns.

Data size. We evaluate CHARM's performance across five graph algorithms and GUPS (§5.1) at different sizes by controlling the number of vertices, ranging from 19 MB (16 vertices) to 5,300 MB (24 vertices). We measure the speedup over RING with 32 (4 chiplets) and 64 (8 chiplets) cores.

As shown in Fig. 10, CHARM consistently outperforms RING across all benchmarks and core counts. Speedup remains stable as graph size increases, indicating that CHARM is influenced more by working set size rather than the total data size. The performance peaks as long as the working set fits within the cache. With 32 cores, CHARM performs best at 79 MB and 320 MB, particularly for SSSP, GUPS, and Graph500, as these sizes align with the L3 cache capacity of AMD EPYC Milan CPUs. With 64 cores, CHARM achieves even greater speedups due to RING's limited scalability and CHARM's balanced task placement across chiplets.

In summary, CHARM's adaptive runtime efficiently manages data placement and scheduling, leveraging partitioned L3 caches while minimizing inter-chiplet data movement. This flexibility enables CHARM to sustain high performance across diverse graph sizes and core counts.

Irregular access patterns. Next, we investigate the impact of irregular data and memory access patterns on CHARM's performance. We evaluate CHARM using a stochastic gradient descent (SGD) algorithm [3] for logistic regression on a dataset with 10,000 samples and 8,192 features, totaling approximately 6,250 MB. To this end, we run the CHARM runtime on top of DimmWitted [51], an analytics engine optimized for statistical computations on modern NUMA architectures (noted as `DW+CHARM`). In addition, we compare the performance of CHARM against four baselines, including native task scheduling schemes of DimmWitted (`DW`):

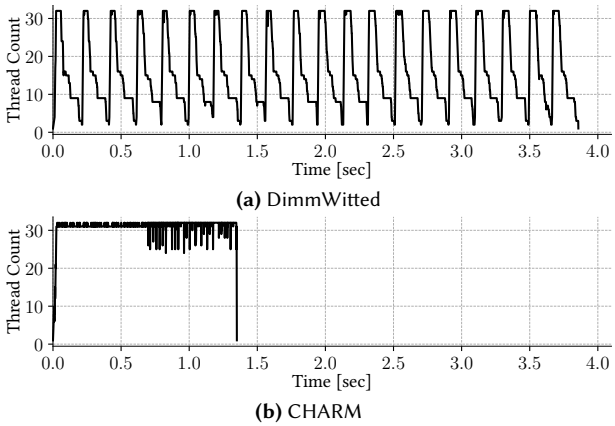
- `DW-per-core`: assigns one task per CPU core, maximizing parallelism;
- `DW-NUMA-node`: allocates one task per NUMA node. It maintains a mutable state for each NUMA node, minimizing inter-node communication; and
- `DW-per-machine`: single task for the machine.

We also use another baseline:

- `DW+CHARM+std::async`: employs OS-level task scheduling to replace CHARM's coroutines using standard C++ `std::async` compiled with GCC 12 and `-O3`.

We measure the throughput of the loss function and gradient calculation of the SGD algorithm for `DimmWitted+CHARM` and all the baselines, with core counts ranging from 8 to 128. Throughout this section, *throughput* denotes the rate at which the SGD kernel processes application data (in GB/s), *not* the sustained DRAM bandwidth of the platform. Each epoch touches the 6 GB input multiple times (loss evaluation and gradient computation), so the logical data volume moved through the compute pipeline far exceeds the dataset's static size. Moreover, the 6 GB working set already surpasses the combined 32 MB $\times 8 = 256$ MB L3 capacity per socket, ensuring sustained pressure on each chiplet's cache and memory subsystem. We validate the same effect at scale with DuckDB in §5.6, which uses an order-of-magnitude larger input.

Fig. 11 shows the results. CHARM, i.e., `DimmWitted+CHARM`, improves the performance of DimmWitted with increasing core counts for the logistic loss calculation (Fig. 11a), with throughput peaking at 165 GB/s. Note that `DimmWitted+CHARM+std::async` results in a notable drop in throughput due to the overhead of `std::async`. Among DimmWitted's



(a) DimmWitted

(b) CHARM

Fig. 12: Thread concurrency during SGD with 32 cores and 10,000 exponents

native strategies, `DimmWitted-NUMA-node` delivers the best performance, reaching 50 GB/s. However, none of the native strategies scale well with increasing core counts.

The same trend is observed in the gradient computation, as shown in Fig. 11b. The throughput almost remains constant for all DimmWitted-native scheduling schemes with increasing core counts. `DimmWitted-NUMA-node` delivers the best performance again, reaching 40 GB/s with 64 cores. The baseline `DimmWitted+CHARM+std::async` performs even worse than `DimmWitted-NUMA-node`, with throughput of 28 GB/s with 64 cores. CHARM (i.e., `DimmWitted+CHARM`) significantly boosts throughput, reaching up to 106 GB/s with 64 cores.

We attribute CHARM's superior performance to two reasons: (i) the chiplet-aware task placement and (ii) the lightweight coroutine implementation. For reason (i), as already analyzed in §5.2 and §5.4, CHARM's global scheduler enhances data locality within chiplets, optimizing cache usage and minimizing main memory access.

We also highlight the significant performance boost from reason (ii)—CHARM's use of coroutines, especially for the gradient computation and model updates. Unlike DimmWitted's reliance on `std::async` by mapping each task to a separate thread, CHARM's coroutines run multiple tasks on a single thread and, therefore, significantly reduce the overhead. Fig. 12 shows the thread concurrency during SGD execution with and without CHARM. In particular, DimmWitted shows an average thread count of 16.2 that fluctuates consistently, resulting in overheads of context switching and poor synchronization. In contrast, Fig. 12b shows a stable thread count for CHARM, with an average of 31.1 due to its controlled manner of concurrency management.

The main limitation of `std::async` is that it blocks threads, often requiring the creation of more threads to manage tasks. In contrast, CHARM uses cooperative multitasking, allowing coroutines to yield without blocking, thus running multiple tasks on the same thread. For example, while DimmWitted creates 641 threads on 32 cores, CHARM used only 34, reducing thread creation overheads. In addition, `std::async` relies on OS-level thread switching, which is slower than CHARM's lightweight user-space context switching, which results in faster task execution.

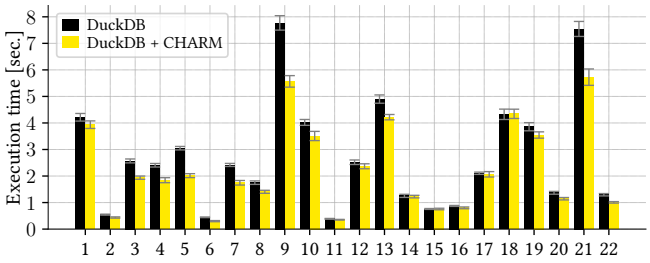


Fig. 13: TPC-H queries on DuckDB

5.6 Performance on OLAP

We evaluate CHARM's impact on OLAP workloads by integrating it into DuckDB [35] by overriding its task scheduling and thread mapping. Using TPC-H at scale factor 100, we compare `DuckDB+CHARM` to unmodified `DuckDB` on 8 cores (equal to one chiplet). The 8-core limit ensures execution time remains long enough to observe CHARM's impact on throughput, as higher core counts reduce the execution time, making it too short for meaningful analysis.

Fig. 13 shows the execution time for each TPC-H query. All queries benefit from CHARM's chiplet-aware runtime, resulting in a reduced execution time with negligible overhead. The most notable gains appear in queries using `hash-joins` and `inner-joins` on large tables (e.g., Q3, Q4, Q5, Q7, Q9, Q10), with speedups ranging from 1.2× to 1.5×. For example, Q21—featuring multiple joins on the `lineitem` table—achieves a 1.3× speedup due to CHARM keeping frequently accessed data within the same chiplet, enhancing cache reuse and reducing traffic.

These gains stem from CHARM's adaptive controller, which distributes threads across chiplets to exploit aggregate L3 cache capacity for large joins (also improving Q8, Q19, Q20). For queries with smaller working sets (e.g., Q1, Q2, Q6, Q11), CHARM compacts threads onto fewer chiplets, minimizing inter-chiplet communication and improving locality. In contrast, Q18—dominated by `hash group-by` operations—shows limited improvement due to uneven data distribution across chiplets, which hampers cache optimizations. As for scalability, increasing the thread count narrows the performance gap between `DuckDB` and `DuckDB+CHARM`, because DuckDB's random distribution eventually utilizes all cores across chiplets.

While CHARM is designed primarily for data-parallel workloads with substantial per-task computation, we also observe benefits at smaller scales, for example, in OLAP queries whose tasks process 2–4 MB of data and run for less than one second. Although we do not define a strict lower bound on task granularity, our results show that short-lived tasks benefit from CHARM's locality-aware scheduling for moderately sized working sets. CHARM's profiling interval is also configurable to adapt to short-lived or fine-grained workloads.

In conclusion, CHARM adjusts thread placement based on runtime behavior and query type: join-heavy queries with large working sets trigger expansion across chiplets for bandwidth and caches; scan/filter-heavy queries benefit from consolidation within a chiplet to preserve locality [9].

5.7 Performance on OLTP

Finally, we investigate the impact of CHARM's chiplet-aware scheduling on database performance, i.e., online transaction processing (OLTP) workloads. We modify ERMIA [19], a memory-optimized OLTP system, to examine the trade-off of the CHARM runtime between maximizing cache locality and increasing available cache size across multiple chiplets. In particular, we adapt ERMIA's scheduling schemes to two distinct policies: `LocalCache` and `DistributedCache`. ERMIA is tightly coupled with its own internal thread management and does not expose sufficient hooks for external schedulers to influence task placement or execution policies. This limits our ability to integrate CHARM's adaptive runtime logic without invasive modifications to the engine's core.

`LocalCache` improves cache locality by limiting operations to cores within a few chiplets, reducing inter-chiplet communication but restricting L3 cache size. In contrast, `DistributedCache` spreads operations across more chiplets, increasing cache capacity but with higher communication overheads. These static policies approximate CHARM's dynamic task mapping, enabling us to assess the benefits of chiplet-aware scheduling without major changes to ERMIA's architecture. We evaluate their performance at different core counts using the YCSB [6] and TPC-C [42] benchmarks.

While this approach does not directly evaluate CHARM's adaptive logic, the experiment remains informative for three reasons: (i) the throughput gap between the two static policies quantifies the intrinsic sensitivity of a real OLTP system to chiplet-level placement. (ii) It establishes a concrete performance envelope: the gap between the two policies reveals the potential optimization headroom, while the stronger-performing policy sets a practical performance target that adaptive schedulers such as CHARM must surpass. (iii) By comparing results across YCSB and TPC-C, it evaluates if a single static policy can be universally optimal, or if the best strategy is fundamentally dependent on the application's specific workload characteristics.

The YCSB workload, with its 45% read and 55% read-modify-write operations on a single table, is well-suited for assessing how frequent read-modify-write operations influence cache locality and communication overhead. In contrast, TPC-C's more complex mix of transactional operations and cross-partition accesses evaluates whether increased cache size can benefit workloads with more diverse access patterns, despite the extra communication overhead.

As shown in Fig. 14, YCSB (Fig. 14a) and TPC-C (Fig. 14b) show nearly identical performance between `LocalCache` and `DistributedCache` scheduling policies across all core

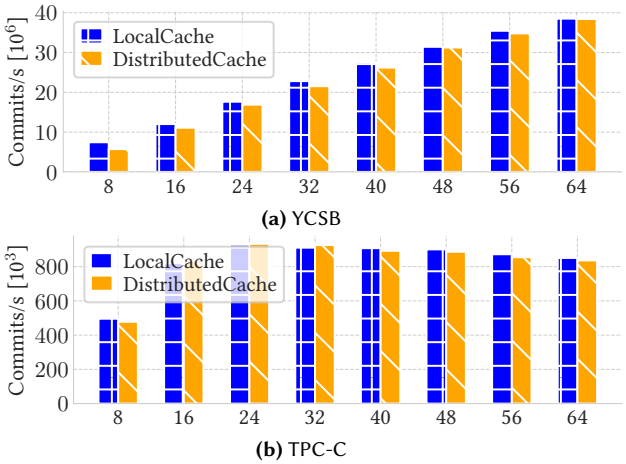


Fig. 14: Commit per seconds for various scheduling policies

counts. This is because OLTP workloads, characterized by short transactions with frequent commits and synchronizations, are less affected by cache locality or larger aggregated cache sizes. Instead, OLTP performance is often limited by commit latency, synchronization overhead, and maintained ACID properties, with frequent inter-thread communication and disk I/O overshadowing cache optimization benefits.

Consequently, our experiment leads to a clear conclusion: for high-contention OLTP workloads such as YCSB and TPC-C, chiplet-level scheduling policies have a negligible impact on performance. The system's sensitivity to cache placement is dwarfed by transactional overheads. This result implies that, for this type of application, there is no workload-dependent optimal policy, because performance is fundamentally indifferent to the underlying chiplet topology. Our findings therefore help delineate the boundaries of applicability for chiplet-aware scheduling, showing that such techniques offer little performance gain for systems bottlenecked by synchronization and commit protocols rather than memory hierarchy effects.

6 Related Work

Profiling-based NUMA-aware schedulers. Modern runtime systems (e.g., SAM [38] and AsymSched [41]) monitor hardware performance to optimize scheduling and placement decisions for multicore architectures. SAM targets traditional multi-socket NUMA systems, aiming to reduce cache and memory contention in multiprogramming; AsymSched focuses on optimizing bandwidth utilization (e.g., hop count) in asymmetric connections. Both, however, fail to fully exploit chiplet-based systems: AsymSched offers limited benefit on chiplet-based designs with uniform interconnects and SAM's profiling events are ill-suited for chiplet-based architectures. In addition, they require centralized and global coordination for placement decisions. In contrast, CHARM uses a decentralized model: each thread independently monitors local and remote memory accesses and dynamically

requests that the global scheduler adjust its placement via the `UpdateLocation` function.

Cache Partitioning. has been widely studied to reduce interference for sharing the LLC. Intel's CAT [15] enables hardware-based partitioning, with further possible improvements to performance isolation and QoS [37, 47], although these rely on static schemes. Dynamic methods such as UCP [34] and TADIP [16] adapt cache management based on application behavior, but overlook the impact of chiplets.

Cache-aware scheduling. Guan et al. explore cache-aware task scheduling for real-time systems, focusing on partitioning caches to reduce interference between tasks in multicore environments [11]; Gracioli et al. review cache management techniques in real-time embedded systems [10]. Recent work has also explored runtime support for emerging chiplet-based architectures. Chirkov et al. evaluate interconnect performance and introduce Meduza, a write-update coherence protocol for chiplet systems [4].

Adaptive scheduling. Existing work has explored adaptive task scheduling in NUMA systems. OmpSs [5] schedules dependent tasks dynamically by prioritizing critical tasks on fast cores. Psaroudakis et al. [33] propose tracking socket-level utilization for adaptive data placement. ATraPos [32] dynamically repartitions OLTP workloads to reduce cross-partition transactions and synchronization.

Chiplet-awareness. Prior work has examined the heterogeneity of inter-core latencies and memory hierarchies in chiplet-based systems [44], including architectural studies of AMD CPUs [29, 39] and energy efficiency improvements [36]. Fogli et al. analyze OLAP performance on chiplet architectures and propose deployment strategies [9], as well as optimizations for high-performance sorting [8].

7 Conclusions

Our research highlights the impact that chiplet-based CPU architectures can have on parallel processing and data-intensive applications. We introduced CHARM, a runtime system that optimizes task allocation and resource management across chiplets. Our evaluation shows that CHARM substantially outperforms state-of-the-art NUMA-aware systems, delivering up to 3.9× speedup in statistical computation, 2.3× in graph processing, and consistently strong performance across a diverse range of memory-intensive workloads.

Our findings are especially beneficial for workloads with irregular access patterns, because they enable the customization of resource allocation policies on novel chiplet-based processors. This work underscores the need for data-intensive systems to move beyond conventional NUMA-aware optimizations, embracing task scheduling approaches that fully exploit the capabilities of modern chiplet-based processors.

Acknowledgments. We sincerely thank our shepherd, Maria Carpen Amarie, and the anonymous EuroSys reviewers for their valuable comments and insightful feedback.

References

[1] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web*. 107–117.

[2] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an China). ACM, New York, NY, USA.

[3] Bob Carpenter. 2008. Lazy Sparse Stochastic Gradient Descent for Regularized Mutlinomial Logistic Regression. https://api.semanticscholar.org/CorpusID:14084261

[4] Grigory Chirkov and David Wentzlaff. 2023. Seizing the Bandwidth Scaling of On-Package Interconnect in a Post-Moore's Law World. *Proceedings of the 37th International Conference on Supercomputing* (2023). https://api.semanticscholar.org/CorpusID:259205930

[5] Kallia Chronaki, Alejandro Rico, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures. *Proceedings of the 29th ACM on International Conference on Supercomputing* (2015). https://api.semanticscholar.org/CorpusID:14318692

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing*. https://api.semanticscholar.org/CorpusID:2589691

[7] Stephane Eranian. 2006. Perfmon2: A Flexible Performance Monitoring Interface for Linux. In *Proceedings of the 2006 Ottawa Linux Symposium*. 269–288.

[8] Alessandro Fogli, Peter Pietzuch, and Jana Giceva. 2024. Optimizing Sorting for Chiplet-Based CPUs. In *Proceedings of the VLDB Workshop*. Imperial College London. https://www.doc.ic.ac.uk/~af6618/files/papers/Optimizing_Sorting_for_Chiplet_Based_CPUs.pdf

[9] Alessandro Fogli, Bo Zhao, Peter Pietzuch, Maximilian Bandle, and Jana Giceva. 2024. OLAP on Modern Chiplet-Based Processors. *Proc. VLDB Endow.* 17 (2024), 3428–3441. https://api.semanticscholar.org/CorpusID:272298657

[10] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. 2015. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Computing Surveys (CSUR)* 48 (2015), 1 – 36. https://api.semanticscholar.org/CorpusID:13709071

[11] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-aware scheduling and analysis for multicores. In *International Conference on Embedded Software*. https://api.semanticscholar.org/CorpusID:3345010

[12] Hacarus. 2024. Analyzing Unconventional Logic Semiconductors – A Shift Away from Semiconductor Manufacturers. https://hacarus.com/ai-lab/03312022-graviton3/. Accessed: 2024-03-01.

[13] Intel. 2024. Accelerating Innovation Through A Standard Chiplet Interface: The Advanced Interface Bus (AIB). https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/accelerating-innovation-through-aib-whitepaper.pdf. Accessed: 2024-05-12.

[14] Intel. 2024. Hardware LLC prefetch feature on 4th Gen Intel® Xeon® Scalable Processor (Codename Sapphire Rapids). https://www.intel.com/content/www/us/en/content-details/780991/hardware-llc-prefetch-feature-on-4th-gen-intel-xeon-scalable-processor-codename-sapphire-rapids.html. Accessed: 2024-05-12.

[15] Intel. 2024. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html. Accessed: 2024-03-01.

[16] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, and Joel Emer. 2008. Adaptive insertion policies for managing shared caches. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 208–219.

[17] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Timothy L. Harris. 2015. Shoal: Smart Allocation and Replication of Memory For Parallel Programs. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:16355072

[18] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware Blocking Synchronization Primitives. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:27706749

[19] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. *Proceedings of the 2016 International Conference on Management of Data* (2016). https://api.semanticscholar.org/CorpusID:15201540

[20] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014). https://api.semanticscholar.org/CorpusID:12770718

[21] Baptiste Lepers and Willy Zwaenepoel. 2023. Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems). In *USENIX Symposium on Operating Systems Design and Implementation*. https://api.semanticscholar.org/CorpusID:259859155

[22] Mian Liao, Daniel H. Zhou, P. Wang, and Minjie Chen. 2023. Power Systems on Chiplet: Inductor-Linked Multi-Output Switched-Capacitor Multi-Rail Power Delivery on Chiplets. *2023 Fourth International Symposium on 3D Power Electronics Integration and Manufacturing (3D-PEIM)* (2023), 1–7. https://api.semanticscholar.org/CorpusID:257260550

[23] Jean-Pierre Lozi, Baptiste Lepers, Justin R. Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux scheduler: a decade of wasted cores. *Proceedings of the Eleventh European Conference on Computer Systems* (2016). https://api.semanticscholar.org/CorpusID:14017684

[24] Michael Mattioli. 2021. Rome to Milan, AMD Continues Its Tour of Italy. *IEEE Micro* 41, 4 (2021), 78–83. https://doi.org/10.1109/MM.2021.3086541

[25] Ke Meng and Guangming Tan. 2017. RING: NUMA-Aware Message-Batching Runtime for Data-Intensive Applications. *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)* (2017), 368–375. https://api.semanticscholar.org/CorpusID:44184357

[26] E. F. Moore. 1959. The shortest path through a maze. In *Proc. of an International Symposium on the Theory of Switching*. Harvard University Press, 285–292.

[27] Richard Murphy, David A. Bader, and et al. 2010. The Graph500 List: Providing Benchmarking Capabilities for Data-Intensive Supercomputing. *Journal of Physics: Conference Series* 78 (2010), 1–10.

[28] Samuel D. Naffziger, Noah Beck, Thomas D. Burd, Kevin M. Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. 2021. Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families : Industrial Product. *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), 57–70. https://api.semanticscholar.org/CorpusID:235415451

[29] Samuel D. Naffziger, Kevin M. Lepak, Milam Paraschou, and Mahesh Subramony. 2020. 2.2 AMD Chiplet Architecture for High-Performance Server and Desktop Products. *2020 IEEE International Solid- State Circuits Conference - (ISSCC)* (2020), 44–45. https://api.semanticscholar.org/CorpusID:215800319

[30] Nevine Nassif, Ashley Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexander M. Kern, William J. Bowhill, David Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad MinHazul Haq, Roopali Sharma, and Kevin Duda. 2022. Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. *2022 IEEE International Solid- State Circuits Conference (ISSCC)* 65 (2022), 44–46. https://api.semanticscholar.org/CorpusID:247523158

[31] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2014. Grappa : A Latency-Tolerant Runtime for Large-Scale Irregular Applications. https://api.semanticscholar.org/CorpusID:13975117

[32] Danica Porobic, Erietta Liarou, Pınar Tözün, and Anastasia Ailamaki. 2014. ATraPos: Adaptive transaction processing on hardware Islands. *2014 IEEE 30th International Conference on Data Engineering* (2014), 688–699. https://api.semanticscholar.org/CorpusID:12976945

[33] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.* 10 (2016), 37–48. https://api.semanticscholar.org/CorpusID:15589515

[34] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)* (2006), 423–432. https://api.semanticscholar.org/CorpusID:9229826

[35] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. *Proceedings of the 2019 International Conference on Management of Data* (2019). https://api.semanticscholar.org/CorpusID:195259571

[36] Robert Schöne, Thomas Ilsche, Mario Bielert, Markus Velten, Markus Schmidl, and Daniel Hackenberg. 2021. Energy Efficiency Aspects of the AMD Zen 2 Architecture. *2021 IEEE International Conference on Cluster Computing (CLUSTER)* (2021), 562–571. https://api.semanticscholar.org/CorpusID:236772121

[37] Vicent Selfa, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit, and María Engracia Gómez. 2017. Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology. *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2017), 194–205. https://api.semanticscholar.org/CorpusID:3474052

[38] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. 2016. Coherence Stalls or Latency Tolerance: Informed CPU Scheduling for Socket and Core Sharing. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:17048199

[39] David Suggs, Mahesh Subramony, and Dan Bouvier. 2020. The AMD "Zen 2" Processor. *IEEE Micro* 40 (2020), 45–52. https://api.semanticscholar.org/CorpusID:214005391

[40] Mikkel Thorup. 1999. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM* 46, 3 (May 1999), 362–394. https://doi.org/10.1145/316542.316548

[41] Christopher Torng, Moyang Wang, and Christopher Batten. 2016. Asymmetry-Aware Work-Stealing Runtimes. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (2016), 40–52. https://api.semanticscholar.org/CorpusID:14372147

[42] Transaction Processing Performance Council. 2010. *TPC Benchmark™ C Standard Specification, Version 5.11.* http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[43] Transaction Processing Performance Council. 2021. *TPC Benchmark™ H Standard Specification, Version 2.17.3.* http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf

[44] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. 2022. Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors. *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering* (2022). https:

//api.semanticscholar.org/CorpusID:247681823

[45] Kefei Wang, Jian Liu, and Feng Chen. 2020. Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores. *Proc. VLDB Endow.* 13, 9 (May 2020), 1540–1554. https://doi.org/10.14778/3397230.3397247

[46] Wei Wang, Jack W. Davidson, and Mary Lou Soffa. 2016. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale NUMA machines. *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), 419–431. https://api.semanticscholar.org/CorpusID:14491359

[47] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. 2018. DCAPS: dynamic cache allocation with partial sharing. *Proceedings of the Thirteenth EuroSys Conference* (2018). https://api.semanticscholar.org/CorpusID:4934627

[48] Haoxuan Xie, Yixiang Fang, Yuyang Xia, Wensheng Luo, and Chenhao Ma. 2023. On Querying Connected Components in Large Temporal Graphs. *Proceedings of the ACM on Management of Data* 1 (2023), 1 – 27. https://api.semanticscholar.org/CorpusID:259213265

[49] Zhuoping Yang, Shixin Ji, Xingzhen Chen, Jinming Zhuang, Weifeng Zhang, Dharmesh Jani, and Peipei Zhou. 2023. Challenges and Opportunities to Enable Large-Scale Computing via Heterogeneous Chiplets. *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)* (2023), 765–770. https://api.semanticscholar.org/CorpusID:265466297

[50] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. 2017. PARSEC3.0: A Multicore Benchmark Suite with Network Stacks and SPLASH-2X. *SIGARCH Comput. Archit. News* 44 (2017), 1–16. https://api.semanticscholar.org/CorpusID:2698588

[51] Ce Zhang and Christopher Ré. 2014. DimmWitted: A Study of Main-Memory Statistical Analytics. *ArXiv* abs/1403.7550 (2014). https://api.semanticscholar.org/CorpusID:475536