StarNUMA: Mitigating NUMA Challenges with Memory Pooling

Albert Cho
Georgia Institute of Technology
Atlanta, GA, USA
acho44@gatech.edu

Alexandros Daglis
Georgia Institute of Technology
Atlanta, GA, USA
alexandros.daglis@cc.gatech.edu

Abstract—**Large multi-socket machines are mission-critical high-performance systems for workloads requiring massive memory shared by hundreds of processors. Beyond eight sockets, such systems typically feature multi-hop inter-socket networks, exacerbating the Non-Uniform Memory Access (NUMA) challenge. NUMA effects stem from major disparity in latency and bandwidth characteristics of local and remote memory, often in the 4–10× range. While judicious data placement across the distributed memory's fragments can ameliorate NUMA effects, we observe that in challenging workloads with irregular access patterns, a large fraction of accessed pages are "vagabond": being actively shared by multiple sockets, they lack a fitting home socket location. On 16-socket systems, such pages incur up to 75% remote memory accesses, which encounter significant latency overheads and bandwidth bottlenecks.**

StarNUMA introduces a new architectural block for multi-socket architectures to ameliorate the challenge posed by vagabond pages. By leveraging the capabilities of the emerging CXL interconnect, StarNUMA augments a typical NUMA architecture with a memory pool that is directly accessible by every socket in a single high-bandwidth interconnect hop. We show that placement of vagabond pages in StarNUMA's memory pool effectively curbs the latency overheads and queuing delays of the bandwidth-constrained multi-hop inter-socket network, reducing the average memory access time of 16-socket systems by 48%**. In turn, faster memory access yields performance improvements of** 1.54× **on average, and up to** 2.17×**.**

I. Introduction

Enterprises employ multi-socket systems for workloads requiring many cores and massive shared memory. While the multi-socket system market is dominated by two- to four-socket machines, there is a niche, yet critically important need for large-scale systems of eight sockets or more. Such mission-critical systems are typically used in high-performance computing (HPC) and transaction processing/banking environments that require thousands of threads with direct access to terabytes of shared memory. Despite driving a small portion of the server market's volume, large-scale multi-socket systems represent a market of $5 billion in annual revenue [59].

Scaling beyond four sockets typically involves hierarchical networks, exacerbating the challenge of Non-Uniform Memory Accesses (NUMA). While every processor can directly access any memory location, memory access *latency* and *bandwidth* characteristics drastically depend on the target memory location's distance from the accessing processor, as a function of the path and hop count taken on the coherent interconnect. In a typical HPE or IBM 16-socket system, the gap between the slowest and fastest memory access exceeds 4×, with an *unloaded* remote memory access latency of up to 360ns [10], [31]. This latency disparity is further exacerbated in a loaded system, as remote memory access is severely bound by inter-socket bandwidth constraints, introducing considerable queuing delays. Techniques that minimize remote memory accesses via intelligent data placement and migrations are therefore essential in large NUMA systems. Unfortunately, some important workloads, such as graphs [3], exhibit challenging irregular access patterns, resulting in a significant fraction of data pages without an evident "home" socket location due to high sharing degree. We call such pages *vagabond.* Vagabond pages hurt performance by incurring many costly remote memory accesses and/or excessive migrations.

To address the challenge of vagabond page placement, we propose the StarNUMA architecture, which extends a typical multi-socket system with a shared memory pool that is directly accessible by every socket in a single high-bandwidth interconnect hop. By leveraging a page hotness monitoring and migration mechanism that considers this new architectural block, vagabond pages can be identified and placed in the pool, thus minimizing costly multi-hop remote memory accesses. Compared to a baseline NUMA system, shared direct access to such a pool provides lower average unloaded latency and higher bandwidth availability for remote memory accesses.

Recent technological trends render StarNUMA a timely and practical solution. The emerging Compute Express Link (CXL) interconnect [18], based on a widely adopted open industry standard, supports all the required features to construct StarNUMA's low-latency and high-bandwidth memory pool. First, CXL supports coherent sharing of disaggregated memory across multiple sockets. Second, for the contained scale of our target multi-socket systems, CXL's performance characteristics allow direct connectivity of every socket to the memory pool, at a 2× lower unloaded latency than the highest multi-hop latency of the baseline system (180ns [38], [46], [56] versus 360ns [10], [31]). Third, CXL is a high-bandwidth interconnect, offering 8GB/s per direction per lane (when operating over PCIe 6.0), with the capability of grouping multiple lanes together at a modest requirement of only four processor pins per lane [2], [56].

By selectively placing vagabond pages in its memory pool, StarNUMA delivers lower unloaded memory access latency and additional bandwidth for remote memory accesses. Our evaluation of a 16-socket system using a variety of workloads shows an overall effective average memory access latency reduction of 48%, yielding an average speedup of 1.54× and up to 2.17×. In summary, we make the following contributions:

- We introduce StarNUMA, a novel NUMA architecture that employs a memory pool to mitigate the performance bottlenecks of high-latency, bandwidth-constrained multi-hop remote memory accesses caused by vagabond pages.
- We propose a practical StarNUMA implementation by leveraging the emerging CXL standard, along with hardware support to facilitate page hotness tracking and migration.
- We construct a novel methodology for scalable and practical microarchitectural simulation of large multi-socket systems.
- We show that StarNUMA accelerates graph, HPC, data serving, and transactional workloads by 1.54× on average and up to 2.17× over a conventional 16-socket system.

Paper outline: §II covers background on large-scale multi-socket systems and CXL, and motivates a memory pool as a new architectural block to mitigate the performance challenges arising from vagabond pages. §III introduces StarNUMA's design, §IV details our methodology and §V evaluates StarNUMA. Finally, §VI covers related work and §VII concludes.

II. Background

A. Large-scale Multi-socket Systems

The building block of modern scalable shared memory architectures is a CPU socket, featuring multiple cores and locally attached DRAM memory. These sockets are interconnected with coherent links—Ultra Path Interconnect (UPI) in Intel's terminology. The key capability enabled by such interconnection is that every processor can directly access any other processor's memory using common load/store instructions. However, a processor's memory access performance depends on whether it accesses local or remote memory, introducing Non-Uniform Memory Access (NUMA) effects.

We study a 16-socket HPE Superdome FLEX [29], [33], [59] as a concrete instance of a large-scale shared memory system; IBM Power10 multi-socket systems are similar [30]. The 16-socket system is organized into four chassis, each housing four sockets, as shown in Fig. 1. The four sockets in the same chassis are directly connected to each other with three UPI links. Some implementations only feature enough UPI links to connect each socket to two other sockets, which results in some pair-wise intra-chassis socket communications requiring two link crossings. We assume there are enough UPI links to allow direct all-to-all intra-chassis socket connection.

All of a chassis' sockets connect to custom inter-socket link ASICs ("FLEX ASIC" for brevity) that provide inter-chassis connectivity [10], [29], [59]. Each FLEX ASIC features enough links to provide direct (i.e., single-hop) connectivity to every other FLEX ASIC in the system. To differentiate from intra-chassis UPI links, we refer to inter-chassis coherent links as NUMALinks, and use the term "coherent links" to collectively refer to both link types.

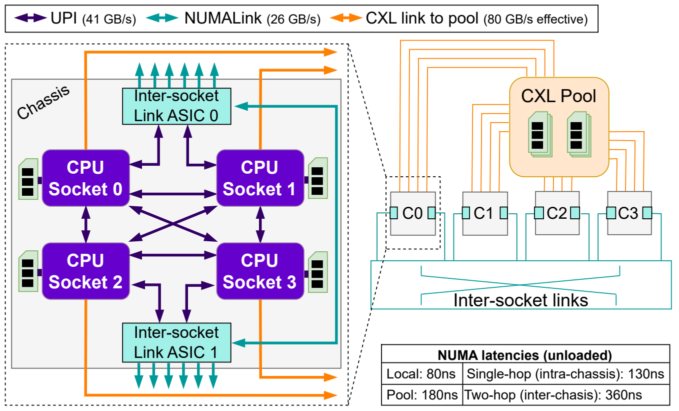


Fig. 1: 16-socket architecture overview consisting of four four-socket chassis. All FLEX ASICs are interconnected with NUMALinks pairwise, connecting any pair of chassis, C0 to C3, in a single hop. The orange square and links annotate StarNUMA's extensions to the baseline multi-socket system.

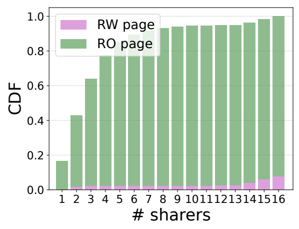
The hierarchical interconnection of the system's 16 sockets reduces the number of required inter-chassis links to 28 (${}_8C_2$ combinations, with two FLEX ASICs per chassis), whereas directly connecting each of the 16 sockets would require 120 links (${}_{16}C_2$ combinations). However, the hierarchical interconnection introduces latency and bandwidth implications that exacerbate NUMA effects.

Intra-chassis memory accesses requiring a single UPI link traversal add 50ns over local memory access. Inter-chassis memory accesses take 360ns [10], [31]. The ∼280ns latency penalty over local memory access includes traversing two UPI links (connecting the socket to the FLEX ASIC at each end), two FLEX ASICs, and an inter-chassis NUMALink, twice. As a result, the unloaded memory access latency in such a 16-socket system is 80ns, 130ns, or 360ns, depending on the target memory's location (local, intra-chassis, or inter-chassis). Bandwidth limitations for cross-socket accesses further aggravate this 4.5× gap in unloaded latency by introducing queuing delays. To illustrate, UPI and NUMALinks have typical bandwidths of 41GB/s and 26GB/s, respectively, while local memory bandwidth usually exceeds 200GB/s.

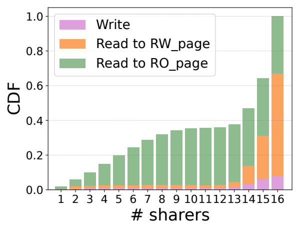
B. The Challenge of Vagabond Page Placement

The wide variance of latency and bandwidth as a function of a memory access' target in large-scale NUMA systems implies that a workload's resulting Average Memory Access Time (AMAT) is significantly affected by its memory access pattern, which in turn directly impacts performance. Therefore, several mechanisms implement intelligent page placement and movement to mitigate the detrimental effect of remote memory accesses that inflate AMAT. However, the most challenging workloads exhibit a significant fraction of memory accesses to vagabond pages, which lack an optimal home socket location.

As an example, Fig. 2 shows page access pattern characteristics for the BFS workload from the GAP graph benchmark suite [11]. The more sharers a page has, the lower its affinity to any specific location. Fig. 2a shows that only 17% of pages



(a) Distribution of page sharing degree (number of sockets with one or more accesses to the page).



(b) Distribution of overall page accesses, classified by accessed page's number of socket sharers.

Fig. 2: Access pattern characteristics for the BFS workload of the GAP benchmark suite on a 16-socket system (1B instr. per core). The distributions account for load/store operations that result in memory access (i.e., LLC miss). Many accesses are concentrated on a small fraction of heavily shared pages.

are accessed by a single socket, while 78% of pages have four sharers or fewer, indicating that an intelligent migration policy may be able to contain page accesses within a chassis and minimize costly 2-hop accesses. However, while only 7% of pages have more than eight sharers (thus no chassis represents a good home location), 68% of all memory accesses are concentrated to those shared pages (Fig. 2b), and the 2% of pages shared by all 16 sockets account for 36% of all accesses. Assuming that these accesses are uniformly distributed across sockets, 75% of them are inter-chassis. We show in §V-A that, even when only unloaded latencies are considered, BFS's access pattern results in a 300ns AMAT. AMAT more than triples when bandwidth constraints of remote memory accesses are also factored in, severely degrading performance. Other workloads we consider exhibit similar behavior, but are omitted here for brevity.

Fig. 2's data suggests that a significant fraction of accesses target few widely shared pages. Explicitly replicating these pages in each socket's memory to increase the fraction of local memory accesses would entail small memory capacity overheads. However, most of these pages are read-write (Fig. 2b), meaning high software complexity and overheads would be required to keep the replicas coherent. Alternatively, a modestly sized memory pool that can host these heavily shared pages and is accessible faster than an inter-chassis memory access can boost performance by accelerating a substantial fraction of memory accesses, while avoiding replication's implications. Hence, we propose *introducing a new placement option for vagabond pages*, by adopting such a memory pool as a new architectural building block for large multi-socket systems.

C. CXL and Memory Pooling

The *Compute Express Link (CXL)* [18] is an open interconnect standard, designed to present a unified solution for coherent accelerators, non-coherent IO devices, and memory expansion devices. It represents the industry's concerted effort for a standardized interconnect that has absorbed a range of competing technologies (OpenCAPI [60], CCIX [19], Gen-Z [24]). Owing to its widespread industry adoption, CXL is bound to become a dominant interconnect, with processor and memory vendors already shaping their products to support it.

Of particular interest in the context of this work is CXL's ability to *disaggregate* memory from the CPU and *share* it across multiple CPUs. In contrast to typical DDR-attached memory, which requires physical CPU proximity to preserve signal integrity, a CXL-attached memory module can be placed at a 20-inch distance, or even further by employing retimers, each adding 20ns roundrip latency overhead [17], [38]. Additionally, such disaggregated memory modules can directly connect to multiple hosts, and support hardware-enforced coherence. Thus, a CXL device placed in the middle of a rack can be deployed as disaggregated memory reachable at low latency (<200ns) by every server in the rack. We call such shared disaggregated memory a *memory pool*.

CXL is also a high-bandwidth interconnect. CXL 3.x is layered over PCIe 6.0, which offers 8GB/s per lane per direction, with the possibility of grouping up to 16 lanes together [20]. For example, an 8-lane (x8) CXL interface offers 64GB/s (40GB/s effective) per direction with modest processor pin requirements (more details in §III-B). Hence, a CXL-attached memory pool can be directly accessible by multiple CPU sockets at both low latency and high bandwidth.

Pond [38] recently demonstrated the flexibility and utility of such a CXL-attached memory pool in the context of a *scale-out* architecture. Aiming to mitigate the challenge of memory stranding, Pond facilitates dynamic allocation of a CXL-enabled memory pool across multiple VM hosts in the same rack. To illustrate, each server in a 16-server Pond group can access the memory pool in 180ns over an x8 CXL link.

Unlike Pond, we propose leveraging memory pooling in the context of *scale-up* (i.e., shared memory) systems, to mitigate the challenge of data placement and NUMA effects. StarNUMA is a multi-socket system augmented with a coherent memory pool that directly connects to every socket with high-bandwidth links. Assuming, like Pond, that such a pool is accessible within 180ns, it offers 40% higher latency than a single-hop remote memory access, but 2$\times$ lower latency than a 2-hop access (see §II-A). Hence, with a placement/migration mechanism that identifies and places vagabond pages in the memory pool, StarNUMA can improve AMAT by converting slow inter-chassis accesses to faster (i.e., low-latency and high-bandwidth) memory pool accesses.

Revisiting Fig. 2's example, of the 36% of memory accesses to pages shared by all 16 sockets, about 75% would be inter-chassis and 25% intra-chassis, assuming they are uniformly distributed. Additionally assuming, for simplicity, that all other memory accesses are local memory accesses, and using the latency values from Fig. 1's table, the resulting AMAT is 160ns ($64\% \times 80ns + 36\% \times (25\% \times 130ns + 75\% \times 360ns)$). With a migration mechanism that identifies the subset of pages shared by all sockets and places them in the memory pool, the latency of inter-chassis accesses can be halved, thus reducing AMAT by 30%, to 112ns. This simplistic first-order estimate ignores any queuing effects due to bandwidth limitations. §V's evaluation shows that the additional bandwidth to remote memory also reduces queuing delays, further reducing AMAT.

III. StarNUMA Design

StarNUMA extends a typical multi-socket system with a CXL-attached memory pool. We base our design on a 16-socket configuration derived from the HPE Superdome FLEX discussed in §II-A. Fig. 1 highlights StarNUMA's extensions over a baseline multi-socket system. The memory pool directly connects over CXL links to all sockets in a star topology, StarNUMA's namesake. Physically, the chassis containing the memory pool would be placed in the middle of the rack, between the four CPU-socket chassis, to minimize its distance from them and hence the number of required CXL retimers, which add latency overhead. Details of the baseline multi-socket system's intra-socket and inter-socket topology, as well as their impact on memory access latency were discussed in §II. We now focus on the design of StarNUMA's newly introduced building block, the CXL-enabled memory pool, and the mechanism for vagabond page placement in the pool.

A. CXL Memory Pool Design Overview

The memory pool is, in CXL terminology, a type-3 Multi Headed Device (MHD) [18] that features multiple CXL ports to support direct connections to every processor socket. As the memory pool is actively shared by all sockets, cache coherence is required, which is supported by CXL 3.x's Back-Invalidate protocol extensions. §III-C further discusses coherence.

We assume the MHD features memory bandwidth and capacity capabilities comparable to a four-socket chassis. We therefore consider a base pool configuration with 16 DDR5 channels providing access to 768GB of memory, representing 20% of the enhanced multi-socket system's total memory capacity. These memory pool characteristics are easily customizable without affecting CPU sockets, which is one of the key strengths of disaggregated memory designs. We consider and evaluate different memory pool capacities in §V-E.

B. Memory Pool Connectivity

Every socket of StarNUMA is directly connected to the memory pool over a dedicated CXL link. As CXL 3.x's underlying physical layer is PCIe 6.0, an 8-lane CXL link provides 64GB/s of raw bandwidth *per direction*. While header and other communication overheads over the CXL port slightly differ depending on the specific access pattern and read/write ratio, we conservatively assume a 62% conversion rate for a realized bandwidth (goodput) of 40GB/s per direction [56].

Eight PCIe 6.0 lanes have a modest requirement of 32 processor pins. To illustrate, a single ECC-enabled DDR4 channel requires over 160 processor pins [32]; DDR5 even more [53]. Hence, the challenge mostly lies in supporting the aggregate number of lanes on the memory pool's MHD: with eight lanes per processor, a 16-socket system would require a total of 128 lanes. As a point of comparison, AMD Zen 4 EPYC CPUs feature an IO die with up to 128 PCIe 5.0 lanes and 12 memory controllers [7]; hence, we expect that the memory pool's MHD can feature similar capabilities.

We derive latency values from Pond's [38] CXL MHD, given its similarities with StarNUMA's memory pool at the hardware level. A distinction that could introduce latency differences is that StarNUMA enforces coherence to implement truly shared memory. The CXL 3.0 whitepaper [18] suggests there is no latency difference between single-owner and multi-owner (where coherence must be enforced) memory segments, likely because a directory lookup is always required, either to check the coherence state, or verify the owner of each memory request (even for single-owner memory segments). Nevertheless, we conservatively include an additional 5ns overhead in the CXL MHD over Pond's latency breakdown.

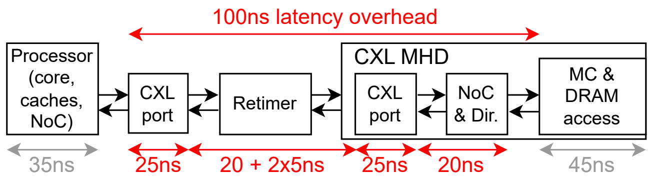


Fig. 3: CXL memory pool access latency breakdown.

Fig. 3 shows StarNUMA's memory pool access latency breakdown. Memory pool accesses traverse the processor's and the MHD's CXL port, each adding a 25ns roundtrip overhead. With 16 sockets, a retimer is likely required between each host and the MHD, adding 20ns, and flight time on the link is about 5ns per direction. Finally, traversing the on-chip network, internal arbitration logic, and coherence directory before reaching the memory controller on the MHD is estimated to be 20ns. Thus, summing up all these latency components, the overhead to access memory at the shared pool is about 100ns. Including on-processor time and DRAM access, the end-to-end unloaded latency of accessing the memory pool is 180ns.

We focus our design on 16-socket systems, which likely represents the ideal scale for a centralized shared memory pool. However, it is possible to scale StarNUMA to 32 sockets and beyond, with the introduction of CXL switches, each adding about 90ns roundtrip latency, for a total memory pool access latency of about 275ns. While the latency gap between a memory pool access and a two-hop NUMA access shrinks, the second advantage of the memory pool, namely additional bandwidth for remotely accessed heavily shared pages, remains. Our design and evaluation focuses on 16-socket systems, but we also perform a latency sensitivity study on larger-scale deployments in §V-C.

C. Cache Coherence

The memory pool must keep cached copies of its address range cached across sockets coherent, as prescribed in the CXL 3.x specification. We assume that the pool implements a directory-based MESI coherence protocol for this purpose. Directory information is distributed across the sockets and memory pool, aligned with the distribution of the address space. Accesses missing in their originating socket are routed to the target address' home node, which initiates all subsequent coherence actions (response/invalidation/forward request). While hardware coherence entails well-known scalability challenges, StarNUMA targets a limited system scale of 8–32 sockets, ameliorating that concern.

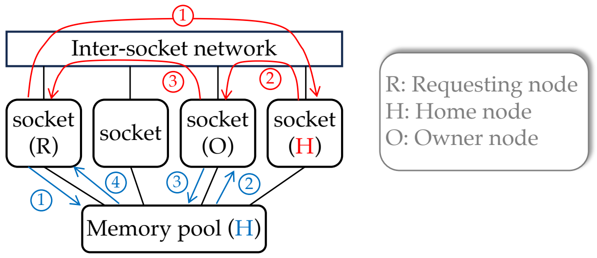


Fig. 4: Two types of coherence-triggered socket-to-socket cache block transfer (3-hop and 4-hop), depending on the home node type (socket vs. pool) of the requested cache block.

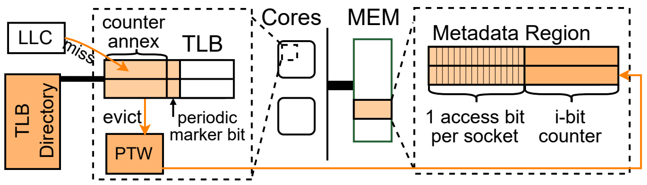


Fig. 5: Hardware support and in-memory metadata for memory access monitoring and page migration in StarNUMA.

StarNUMA embodies two different mechanisms for socket-to-socket block transfers, depending on the type of the target block's home node. If the home node is a normal socket, then the block transfer is completed via the typical 3-hop cache-to-cache transfer optimization (red R→H→O→R path in Fig. 4). If the home node is the pool, block transfers complete in four hops via the pool (blue R→H→O→H→R path). Counter-intuitively, traversing two CXL links instead of performing a 3-hop block transfer is, on average, faster than directly sending the requested block from the owner to the requesting socket, due to the high latency of cross-chassis link traversals. On our target 16-socket system, the average (unloaded) 3-hop cache block transfer latency is 333ns, derived by averaging the cumulative latency of the three traversed links for all possible R, H, O socket combinations. In contrast, the latency of a 4-hop transfer via the pool—which entails two roundtrips over two CXL links—is only 200ns.

D. Memory Access Monitoring and Page Migration

To effectively leverage StarNUMA, a lightweight mechanism that monitors memory access patterns and successfully identifies vagabond pages for pool placement is essential. Support for page access pattern detection is useful even in baseline NUMA and tiered-memory systems, and is a challenging ongoing research topic [6], [21], [34], [40], [46], [48].

Most prior solutions are software-based and leverage OS-driven page hotness tracking and migration mechanisms. We devise our own mechanism for StarNUMA, for two reasons. First, page sharing degree is a key metric that must be considered to drive effective migrations to the new shared memory pool component. Second, we found that at the data migration rate StarNUMA requires to be effective, conventional software-based mechanisms incur prohibitive overheads; hence, hardware support for this functionality is required. We therefore develop a policy and hardware-supported mechanism to drive page migrations in StarNUMA.

A migration mechanism can be broadly divided into three components: access tracking, migration candidate selection, and migration itself. We discuss the constraints and hardware support required to avoid prohibitive overheads at each stage, and describe the mechanism applied in StarNUMA. Fig. 5 summarizes our extensions for memory access monitoring.

1) Access Tracking: The first component of any migration mechanism is memory access monitoring. In software-based variants, the OS periodically "poisons" a set of selected pages to trigger OS bookkeeping logic via subsequent minor page faults. Monitoring a small page sample is essential to keep the minor page fault overhead low, but we found that practical sample sizes are not large enough to identify pages that would benefit from pool placement at a sufficiently high rate. We therefore devise an approach to make monitoring of the entire address space practical.

We logically split the physical memory into regions, each consisting of several consecutive physical pages—we discuss sizing in §III-D4. We allocate a physically contiguous metadata region in memory to maintain per-region access metadata. Each region's corresponding access tracker entry is used to maintain access statistics and is trivially identified in the metadata region in a base + offset fashion, using region id $\times$ tracker entry size as offset. A tracker design T_i has entries comprising (i) one bit per socket, and (ii) an $i-$bit counter to record up to $2^i - 1$ total region accesses, as described next.

Similar to prior work [41], [47], StarNUMA extends each TLB with a counter annex, as shown in Fig. 5. Briefly, for a T_i tracker design, each TLB entry has an associated i-bit counter. Upon the completion of an LLC-missing load, the corresponding TLB annex entry's counter is incremented. When a TLB entry is evicted, the hardware Page Table Walker (PTW) adds the annex entry's value in the memory's metadata region respective counter. To capture the counters of hot pages that are never evicted from the TLB, all TLB annex entries also have a marker bit, periodically set once per migration phase (about once per second in our implementation). When a TLB entry with a marker set is accessed, the PTW adds the annex entry's value to the respective metadata region location and resets the marker.

A special case of T_i we consider is T_0, which can only track whether a socket has accessed a region during a migration phase, but cannot rank the relative hotness of different regions. T_0 is an interesting design point because it collects sufficient information to identify widely shared pages—which are good candidates for pool placement in StarNUMA—while eschewing the need for value additions, both in the TLB annex and in the metadata region by the PTW.

2) Migration Candidate Selection: While the PTW mechanism continuously updates the access information in the metadata region, the selection of regions to be migrated is done by an OS thread once per migration phase (i.e., about once per second). Algorithm 1 describes our threshold-based decision-making logic that selects the memory regions to be migrated and their destination (another socket or the memory pool).

Briefly, if a region's access count exceeds a set threshold, it is selected to be migrated, either to the pool, or to another socket, depending on its sharing degree. If the destination (including the pool) is out of usable capacity, a victim region is first identified to be moved out, using a similar threshold-based approach. The victim's destination is randomly selected among the region's sharers. Note that the threshold-based approach enables decision-making with a single pass of the metadata region and no sorting requirement. The once-per-phase metadata region scan also resets all page access counters.

Algorithm 1 Migration decision pseudocode

```
 1: Region_Trackers[]
 2: N_MIGRATION ← 0; MIGRATION_LIMIT;
 3: ACCESS_THRES_HI; ACCESS_THRES_LO;
 4: for region in : Region_Trackers do
 5:     # Identify migration candidates:
 6:     if region.accesses ≥ ACCESS_THRES_HI then
 7:         best_location ← random(region.sharers)
 8:         if count(region.sharers) ≥ 8 then
 9:             best_location ← memory pool
10:         end if
11:         if best_location ≠ curr_location then
12:             if region not ping-ponging* then
13:                 # Identify eviction candidate:
14:                 if no space in best_location then
15:                     for victim_region in : Region_Trackers do
16:                         if victim_region.location = best_location AND
                            victim_region.accesses ≤ ACCESS_THRES_LO then
17:                             victim ← victim_region
18:                             BREAK; # found victim_region to move
19:                         end if
20:                     end for
21:                     migrate victim to random(victim_region.sharers)
22:                 end if
23:                 # Perform migration:
24:                 migrate region to best_location
25:                 N_MIGRATION++;
26:             end if
27:         end if
28:     end if
29:     if N_MIGRATION ≥ MIGRATION_LIMIT then
30:         BREAK; # done for this migration phase
31:     end if
32: end for
```

*We consider a region ping-ponging if it has migrated for more than a quarter of the current phase number.

3) Migration: After the regions to be migrated have been identified, they must be moved to the target destination and page mappings must be updated, which involves page table updates and costly TLB shootdowns. Using conventional TLB shootdowns requires an inter-processor interrupt from the core initiating the migration to every other core in the system, for every page migrated. Cores receiving TLB shootdown incur several thousand cycles just to enter kernel mode, even when the TLB entry being shot down is not present in the receiving core's TLB [64]. This overhead gets prohibitive as we increase the number of pages migrated and the system's core count.

To maximize StarNUMA's potential, it is necessary to address the TLB shootdown overhead with a scalable mechanism. Default software-based TLB shootdowns involve costly inter-processor interrupts and execution of kernel handlers, incurring an overhead of thousands of cycles per shootdown, potentially on every core of the system. We therefore adopt hardware support from prior work [64], which introduces a shared TLB directory that allows TLB shootdowns to be sent to only the necessary cores that actually cache a translation of the migrating page in their TLB. Our adopted design [64] also includes minimal core extensions to allow handling TLB entry invalidations entirely in hardware, without OS intervention. The core in charge of orchestrating the migration still incurs the associated overheads of initiating TLB shootdowns and waiting for their completion. Other proposals that drastically improve TLB shootdown scalability via hardware support or batching [8] could also be applicable.

4) Region Sizing: Memory region sizing is an important design knob that dictates several system aspects: (i) the size of the metadata region; (ii) the time required to scan the metadata region (Algorithm 1); and (iii) the hotness tracking precision, which affects the performed migrations' quality. A smaller region benefits (iii) at the cost of higher overheads for (i) and (ii). Assuming a target full-scale system with 16TB of memory and a region size of 512KB, the metadata region comprises 32 million tracker entries. For a 16-socket system and T_{16}, the metadata region is 128MB. At that size, we profile Algorithm 1's runtime and find a total min/max runtime of 64/320 million cycles, depending on the latency of accessing the metadata memory region. This cost falls well within the period of migration decisions in StarNUMA, which is at least one billion cycles (more details in §IV-C).

IV. Evaluation Methodology

We evaluate the baseline system and StarNUMA in simulation. To make simulations of the target system scale and necessary simulated runtime feasible, we construct a novel evaluation methodology by employing a multi-step sampling-based approach (§IV-A) and mixed modality simulation (§IV-B). The latter technique entails modeling sockets of the evaluated system at two different levels of detail; hence we differentiate between "detailed" and "light" sockets.

A. Multi-Step Sampling-Based Simulation

Evaluating multi-socket workloads long enough to capture several page monitoring and migration intervals requires capturing tens of billions of instructions per core. Simulating at such scale and at cycle level requires prohibitive runtimes and resources. To address this challenge, we tailor a sampling-based approach inspired by the SMARTS methodology [65] to the inherent characteristics of our target system, to effectively capture the effects of data placement and migrations.

Fig. 6 shows an overview of the three steps comprising our sampling-based simulation. In step A, we collect instruction and memory traces of our target workloads on real hardware. Step B feeds the memory traces into a memory trace simulator, which makes data migration decisions at time intervals typical of modern systems. Step C simulates each of these intervals along with its associated data movement decisions at cycle level. The three steps are detailed next.

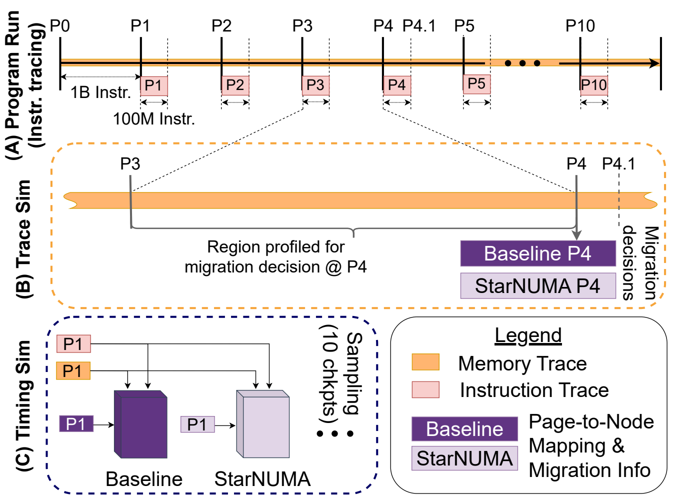


Fig. 6: Evaluation methodology flow for a workload that executes 11 billion instructions per core.

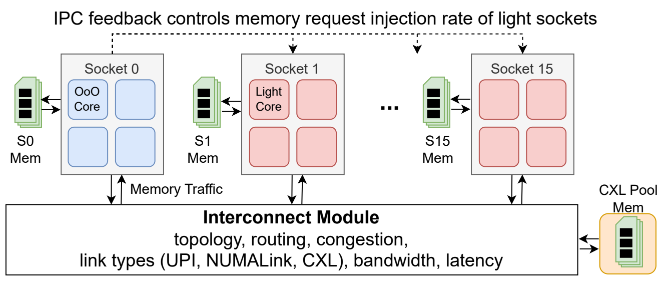

IPC feedback controls memory request injection rate of light sockets
Socket 0
OoO Core
Socket 1
Light Core
Socket 15
S0 Mem
S1 Mem
S15 Mem
Memory Traffic
Interconnect Module
topology, routing, congestion,
link types (UPI, NUMALink, CXL), bandwidth, latency
CXL Pool Mem

Fig. 7: Overview of mixed-modality simulation.

1) Step A – Instruction and Memory Access Tracing: We deploy the target workload on a real machine and trace each application thread's execution. In the same multi-threaded program run, the tracer records instruction traces for the threads that will be simulated on the detailed socket's cores, and, in separate files, the memory accesses of *all* threads (including those mapped to the "light" sockets), each tagged with its corresponding dynamic instruction count. The memory traces are processed in step B by the trace simulator to inform migration decisions, and are also used in step C to generate the memory access traffic of light sockets during timing simulation. We evaluate a 16-socket system with scaled-down 4-core sockets, hence we trace 64-thread program runs.

Our tracer combines ChampSim's tracer [1] and an existing Memory Access tracer [23], both based on Intel Pintool [42]. We record memory traces at one billion instruction intervals, which we refer to as a *phase*. We also record the first 100 million instructions of each phase, as shown in Fig. 6(A). *All instruction counts mentioned throughout §IV are per thread.*

2) Step B – Memory Trace Simulation: This step processes only the memory traces to make per-phase data migration decisions according to the implemented page monitoring and migration policy, as later described in §IV-C. The output of this trace-based simulation is a set of *checkpoints* containing the page-to-socket mapping at the end of each phase as well as a list of migrations that should occur in the upcoming phase. Each checkpoint is used to initiate a timing simulation (step C). The memory state at the start of phase P_N is the result of cumulative migration decisions of the preceding N-1 phases, and the Nth checkpoint indicates the set of migrations that must be modeled during phase P_N's simulation.

3) Step C – Timing Simulation: Our timing simulation model is based on ChampSim [1]. We model a 16-socket system by employing §IV-B's mixed-modality approach. The simulation comprises N parallel timing simulations, one per generated checkpoint $P_{1..N}$. The inputs of each timing simulation are the phase's corresponding memory and instruction traces collected in step A, and memory map and migration decisions generated in step B (as shown in Fig. 6(C) for P_1).

We use 5–10 checkpoints per workload, a billion instructions apart, and simulate 100 million instructions (per thread) at each checkpoint, for a total of 32–64 billion simulated instructions per workload (of which 2–4 billion are executed on detailed cores). Each timing simulation is primed with a warm-up phase long enough to populate each socket's LLC; we find 10–20 million instructions sufficient for all our workloads. A workload's reported statistics are derived by aggregating results across the simulation of all its checkpoints.

B. Mixed-Modality Simulation

Simulating an entire multi-socket system at cycle level remains impractical even with a sampling-based approach. We therefore scale down the socket size, i.e., the number of simulated cores per socket, as shown in Fig. 7. Additionally, we simulate one socket in full microarchitectural detail and the remaining sockets as simpler endpoints ("light" sockets).

Each light core injects its own unique memory trace collected in step A (§IV-A1) at an injection rate regulated by using the measured IPC of the socket modeled in detail, assuming all threads of the same workload achieve, on average, similar IPC. Each light socket features an LLC-sized cache to support coherence modeling and filter accesses to memory, as well as a detailed memory controller model to service memory requests that target its memory range, while accurately capturing performance effects of memory scheduling and contention.

An interconnect simulation module between the LLC and the memory controller of each processor models the inter-socket topology, along with its associated link latencies and bandwidth limitations, thus capturing contention/queuing effects. Upon a new memory access request, the destination socket is determined by looking up the page map input generated by step B. The interconnect module then determines the sequence of links the request must traverse, based on its source and destination socket. After traversing the interconnect, the request either enters the destination socket's memory controller queue, or triggers coherence events as needed.

C. Migration Overhead Modeling

We model §III-D's migration mechanism using 512KB regions (i.e., 128 4KB pages per region). We evaluate StarNUMA for T_{16} and T_0. We find an effective HI threshold for T_{16} to be 20K∼400K region accesses. Thus, we start with a HI threshold of 20K and dynamically adjust it in every migration phase as a simple function of page count exceeding the threshold relative to the set migration limit. We experiment with 0 to 256K 4KB pages as Algorithm 1's per-phase migration limit, choosing the best-performing limit for each workload-system combination. Similarly, the eviction (LO) threshold starts at 1K and is adjusted dynamically (up to 10K in our experiments). T_0 uses a fixed HI threshold of 16 (i.e., only consider regions touched by all sockets).

Since we provision hardware support for page access monitoring and TLB shootdowns, most of the overheads that are typically associated with the OS are offloaded to hardware, which also facilitates the use of userspace simulation for accurate performance evaluation. The only overhead component remaining in the OS is the metadata region scanning and page table updates. Given the work required per migration period, one dedicated core is sufficient (see §III-D4), and is a negligible overhead for a system of our target scale (e.g., 0.2% for a 16-socket system with 448 cores—see §IV-D). As we adopt hardware-supported TLB shootdowns, victim cores do not incur expensive OS-driven TLB shootdown cost; the migration-initiating core incurs a 3k-cycle cost per page migration [64]. TLB shootdowns still invalidate TLB entries as needed and TLB misses trigger page walks.

We model the additional memory traffic required for tracker updates, as well as the inter-socket data movement triggered from migration decisions. While a page's migration is in flight, all memory accesses to that page are stalled until the migration completes. Initial page placement is determined using a first-touch policy. At each migration phase, a list of pages to be migrated is determined based on Algorithm 1. While the entirety of these migrations is performed in the trace simulation (step B—§IV-A2), cycle-level timing simulation (step C—§IV-A3) only covers the first 100 million instructions of each billion-instruction phase. Hence, only the first 10% migrations of each phase are modeled during timing simulation.

Baseline System Migration: Our migration policy is tailored to primarily identify good candidates for pool placement. To highlight the significance of the pool as a new building block for multi-socket systems rather than the specific migration policy implemented, we favor the baseline by assuming *zero-cost per-socket* knowledge of *all* accesses to *every* 4KB page at each migration interval. After zero-cost migration decisions based on complete page access knowledge are made, the migration cost itself is modeled as in StarNUMA.

D. System Configurations

Table I summarizes the key parameters of our target full-scale 16-socket system, modeled after an HPE Superdome Flex configuration [31]. To make simulation practical, we scale down the system to four cores per socket, adjusting the rest of the system's parameters accordingly. The number of memory channels on each socket and the pool (thus their capacity and bandwidth), as well as the bandwidth of the UPI, NUMA, and CXL links are commensurately scaled down. Table II summarizes the scaled-down simulation parameters.

TABLE I: System parameters of full-scale baseline 16-socket system and StarNUMA. See the system layout in Fig. 1.

CPU socket	28 OoO cores, 2.4GHz, 4-wide, 256-entry ROB
L1	32KB L1-I & L1-D, 8-way, 64B blocks, 4-cycle access
L2	1 MB, 16-way, 14-cycle access
LLC	2MB/core, 16-way, 30-cycle access, shared (per socket) & non-inclusive
Memory	DDR5-4800, 32 GB per channel, 6 channels per socket
Network Topology	4 chassis $\times$ 4 sockets per chassis (see §II-A) Hierarchical inter-socket interconnect (see Fig. 1) ▷ Intra-chassis: all-to-all socket-to-socket UPI ▷ Inter-chassis: all-to-all chassis-to-chassis NUMALink
Link Bandwidth (per direction)	20.8GB/s per UPI link (4 links per socket) 13GB/s per NUMALink (12 links per chassis)
Remote Access Latency Penalty	50ns (within chassis group), 280ns (inter-chassis) (breakdown in §II-A)
Memory pool	
Memory	DDR5-4800, 48 GB per channel, 16 channels
Pool Bandwidth	x8 CXL port per socket (direct per-socket link) ⇒ 64GB/s raw (40GB/s effective) per direction
Latency Penalty	100ns (see §II-C)

TABLE II: System parameters used for simulation on ChampSim. The table only shows parameters that differ from the full-scale system (Table I).

CPU socket	4 cores, core microarchitecture unchanged
Memory	1 DDR5-4800 channel per socket
Link Bandwidth	3GB/s per direction for each UPI link or NUMALink
Memory pool	
Memory	2 DDR5-4800 channels
Pool Bandwidth	6GB/s per direction supported from each socket

As described in §III-A, we consider a memory pool capacity equivalent to a four-socket chassis' aggregate memory. However, the memory footprints of our workload instances are naturally dwarfed by the footprint of real 16-socket deployments. Hence, instead of imposing an absolute pool capacity limit, we limit the amount of data allowable on the pool as 20% of the memory used by each workload.

E. Workloads

We use workloads from four application families:

- *Graph Analytics:* We deploy four graph analytics workloads from the widely used GAP benchmark suite [11]: Breadth-First Search (BFS), Connected Components (CC), Single-Source Shortest Paths (SSSP), and Triangle Counting (TC), all operating on a Kronecker graph with 2^{28} vertices and an average degree of 32, requiring ∼50GB of memory.
- *HPC Workloads:* We deploy two genomics analysis pipelines from GenomicsBench [57]: Full-Text Index in Minute Space (FMI) and Partial-Order Alignment (POA). The memory footprint of these workloads is ∼10GB.
- *Data Serving:* We use the Masstree high-performance Key-Value store [44] with a 100GB dataset, uniform key popularity distribution, and 50/50 read/write ratio.

TABLE III: Workload summary. Numbers in parentheses show the per-core IPC when the workload runs on a single socket.

Wkld	IPC	LLC MPKI	Wkld	IPC	LLC MPKI
SSSP	0.06 (0.56)	73	Masstree	0.18 (0.89)	15
BFS	0.10 (0.69)	32	TPCC	0.41 (1.12)	4.8
CC	0.14 (0.78)	17	FMI	0.61 (1.45)	2.6
TC	0.40 (1.7)	3.2	POA	0.68 (0.68)	33

- *Transaction Processing:* We deploy TPCC with 64 warehouses on the Silo in-memory DBMS [52], [61]. TPCC's resulting memory footprint is 12GB.

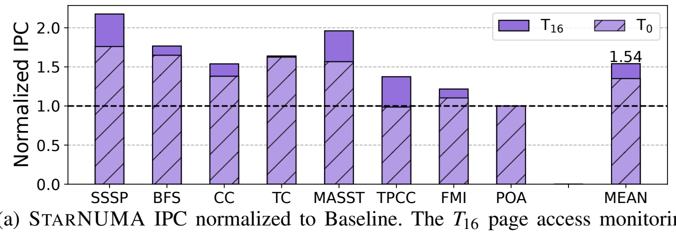
Table III summarizes our workloads' per-core IPC and LLC MPKI as measured on the baseline 16-socket system. Numbers in parentheses show the per-core IPC achieved by single-socket execution with local memory only. The 2–10× IPC gap between single- and 16-socket execution illustrates the performance impact of NUMA effects.

V. Evaluation Results

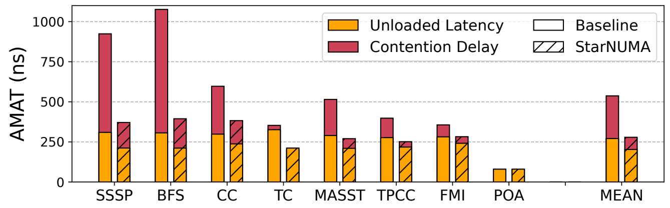
A. Main Results

We start by comparing StarNUMA with a T_{16} region monitoring mechanism against the baseline multi-socket system that uses perfect page access knowledge (see §IV-C). Fig. 8c shows the breakdown of memory accesses according to their type. Fig. 8b shows the measured AMAT, decomposed into unloaded latency and delay from contention at inter-socket and CXL links. *Unloaded Latency* is the expected AMAT without any contention, and the fraction of the measured latency it corresponds to is analytically derived from Fig. 8c's breakdown as $\sum(\%\ access\ type) \times (access\ type's\ unloaded\ latency)$ across all access types: local (80ns), 1-hop (130ns), 2-hop (360ns), pool accesses (180ns), and coherence-triggered block transfers (BT). For BT, we account 413ns for socket-to-socket transfers (BT_Socket) and 280ns for block transfers via the pool (BT_Pool), derived as the required network traversal latency in each case (see §III-C) plus 80ns for memory access and directory lookup. *Contention Delay* in Fig. 8b is the difference between the *measured* AMAT and our *computed* unloaded latency. Finally, Fig. 8a shows StarNUMA's speedup over the baseline, as a result of the achieved AMAT reduction.

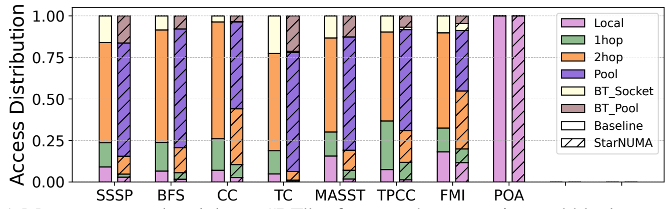
StarNUMA (with T_{16}) reduces AMAT by 48% on average, resulting in 1.54× speedup. TC and FMI are compute-intensive with low LLC MPKI (Table III); hence, they experience only minor contention in the baseline and most of StarNUMA's speedup is attributed to lower-latency access to shared data provided by the pool. In contrast, SSSP and BFS are bandwidth-bound, with their AMAT largely determined by contention in the baseline. StarNUMA reduces latency in two distinct ways: (i) by directly reducing unloaded latency (fewer hops on average); (ii) by reducing contention, which in turn mitigates queuing delay. A big fraction of StarNUMA's achieved AMAT reduction stems from (ii), as StarNUMA leverages the CXL links' additional bandwidth to heavily shared pages, mitigating the queuing delay for remote memory accesses. For the remaining workloads (CC, Masstree, TPCC), the pool's high bandwidth and low latency jointly contribute to AMAT reduction and the resulting speedup. Lastly, POA is completely insensitive to NUMA effects: all accesses are local, hence no migration occurs, and no data is placed in the pool. POA represents workloads with very localized accesses where a simple first-touch page placement policy alone prevents all detrimental NUMA effects.



(a) StarNUMA IPC normalized to Baseline. The T_{16} page access monitoring mechanism achieves additional speedup over the simpler T_0.



(b) Average Memory Access Time (AMAT). StarNUMA results refer to T_{16}.



(c) Memory access breakdown. "BT" refers to coherence-triggered block transfer and StarNUMA results refer to T_{16}.

Fig. 8: Speedup, Average Memory Access Time (AMAT), and memory access breakdown of StarNUMA versus baseline.

TABLE IV: Fraction of migrations to the pool.

Workload	Migration to pool
SSSP	80%
BFS	100%
CC	99%
TC	80%
Masstree	100%
TPCC	93%
FMI	47%
POA	0%

Coherence activity (BT) represents ∼10% of memory accesses in most cases. Most BT in StarNUMA complete via the pool path, which is 30% faster than 3-hop inter-socket transfers on average. Overall, the fraction of AMAT reduction attributed to faster BT is minimal. While coherence traffic does not become a bottleneck, it is still commonly occurring: the CXL directory handles a coherence transaction every 100ns on average, an unsustainable frequency for software-based coherence.

Finally, we provide some additional results to underline the significance of the pool as a new data placement option. First, Table IV shows the fraction of migrated pages StarNUMA moves to the pool. Excluding POA, the geomean of the fraction of migrations to the pool is 83%, with several workloads at 90% or higher, indicating that most heavily accessed regions are also widely shared (partially a side-effect of the large region size used). Socket-to-socket transfers are

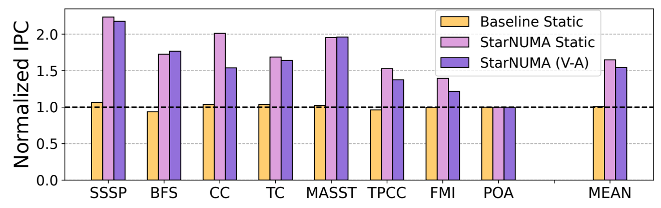


Fig. 9: Speedup using initial static page placement with oracular knowledge, normalized to baseline with dynamic migration.

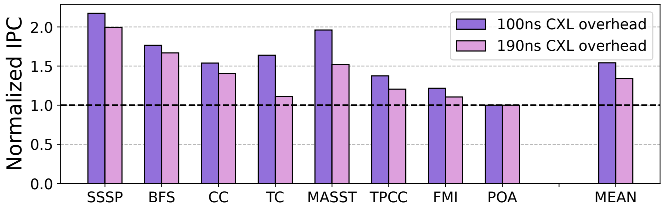


Fig. 10: Speedup over baseline for different CXL latency penalties, affecting memory pool access latency.

few also because the first-touch policy properly places private pages from the very beginning. Second, Fig. 8a also shows the achieved speedup with the simpler T_0 mechanism. Unlike T_{16}, T_0 cannot identify the hottest regions, but is enough to identify highly shared regions that would benefit from pool placement, thus capturing most of T_{16}'s gains for a 1.35× speedup. The rest of the evaluation uses StarNUMA with T_{16}.

B. Comparison to Oracular Static Page Placement

To provide deeper insight on the significance of the memory pool as a building block versus the specific migration policy we evaluated, we also consider a static initial page placement (i.e., no dynamic data movement at runtime) using oracular a priori knowledge of each workload's access pattern. Fig. 9 shows the results when such static placement is applied to both the baseline and StarNUMA architectures, compared against StarNUMA with dynamic migration from §V-A.

Static oracular placement slightly outperforms the dynamic migration alternative for StarNUMA, as it eliminates migration overheads. The result indicates that sharing patterns do not drastically change over time, hence simpler migration support to reap StarNUMA's benefits may be possible. Most importantly, the baseline with static placement does not yield any gains over the baseline with first touch placement + dynamic migration. This result strongly underlines our key observation that *baseline NUMA systems architecturally lack a good location for vagabond pages*, highlighting StarNUMA's memory pool as a crucial new building block.

C. Impact of Memory Pool Latency

Fig. 10 evaluates StarNUMA's sensitivity to the memory pool access latency. In addition to default StarNUMA's 100ns latency overhead, we consider a 190ns overhead. This value represents the addition of an intermediate CXL switch (§III-B), resulting in an unloaded end-to-end memory pool access latency of 270ns—still 25% lower than a 2-hop access.

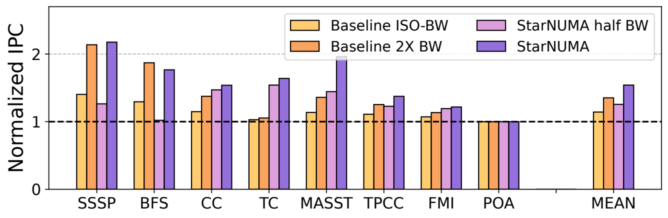


Fig. 11: Speedup over baseline multi-socket system for system configurations with different link bandwidth provisioning.

The increased latency is detrimental to most workloads. On average, the higher latency overhead reduces StarNUMA's speedup over the baseline from 1.54× to 1.34×. TC is the workload most affected, with a speedup drop from 1.63× to 1.11×, because its speedup almost exclusively stems from the latency reduction afforded by StarNUMA, as seen in §V-A.

D. Impact of Bandwidth Availability

§V-A reveals that bandwidth availability critically affects the performance of several workloads. We explore three additional system configurations to analyze bandwidth's impact further:

- **Baseline ISO-BW:** Augment the bandwidth of coherent links by an aggregate amount matching the 640GB/s effective link bandwidth added by StarNUMA. By pro-rating this amount based on each link's base bandwidth, we assume 26.4GB/s UPI and 17GB/s NUMALink (per direction, up from 20.8GB/s and 13GB/s, respectively).
- **Baseline 2×BW:** Augment the bandwidth of every coherent link by 2× to evaluate the gains attainable via brute-force bandwidth overprovisioning, without pool-afforded latency gains. The aggregate bandwidth overprovisioning is much higher than the additional bandwidth of StarNUMA's 16 CXL links. The 16-socket system features a total of 68 coherent links (28 inter-chassis and 40 intra-chassis), thus doubling their capability boosts the baseline's aggregate bandwidth by 1.2TB/s. Such overprovisioning is impractical, as it also doubles processor pin requirements.
- **StarNUMA Half-BW:** Halve CXL link bandwidth to 20GB/s (i.e., scale down x8 CXL links to x4).

Fig. 11 shows the performance of these configurations plus default StarNUMA from §V-A, normalized to the baseline multi-socket system. StarNUMA still outperforms the impractically overprovisioned Baseline 2×BW by 12% on average. Only for the most bandwidth-bound workload, BFS, Baseline 2×BW slightly outperforms StarNUMA, not only because of its aggregate bandwidth superiority, but also because it utilizes its bandwidth resources more uniformly: The baseline spreads accesses to heavily shared pages uniformly across inter- and intra-chassis links. In contrast, StarNUMA aggressively migrates the hottest shared pages to the pool (Table IV), thus a high fraction of memory accesses is concentrated on the pool's CXL links, which get highly contended while inter-socket links remain highly underutilized.

Baseline ISO-BW outperforms the baseline by 1.14×, with the highest gains being 1.4× and 1.29× for the bandwidth-bound SSSP and BFS, respectively. Still, it trails behind

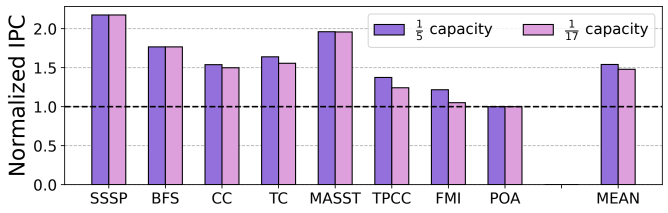
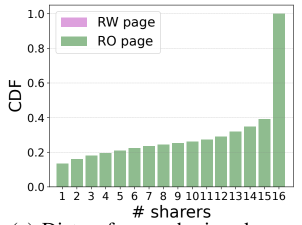
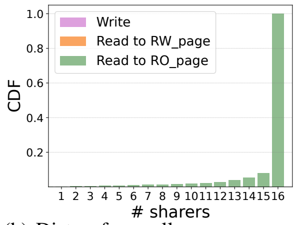


Fig. 12: Speedup over baseline for different memory pool capacities. $\frac{1}{17}$ and $\frac{1}{5}$ indicate memory pool capacity equivalent to a socket and a chassis (i.e., four sockets), respectively.



(a) Distr. of page sharing degree.



(b) Distr. of overall page accesses.

Fig. 13: Access pattern characteristics for TC.

StarNUMA by 40%. StarNUMA Half-BW still outperforms Baseline ISO-BW by 11% on average. Despite halving bandwidth to the pool, StarNUMA Half-BW achieves 1.21× speedup for SSSP, but only 2% for BFS, as it experiences the bottleneck of all traffic to shared data concentrating on the pool's links, as described above. In summary, StarNUMA's achieved reduction in unloaded latency plays a drastic role in reducing AMAT and improving performance. Boosting a conventional multi-socket system's bandwidth is *neither necessary nor sufficient* to approach the performance gains attainable via StarNUMA's shared memory pool.

E. Impact of Memory Pool Capacity

So far we assumed a memory pool with capacity equivalent to one chassis' (i.e., four sockets) worth of memory, representing 20% of the system's total memory capacity. We now examine StarNUMA's performance with a pool capacity equivalent to a single socket's. Thus, we reduce the pool's capacity from 20% to $\frac{1}{17}$ of each workload's memory footprint.

Fig. 12 shows StarNUMA's performance sensitivity to memory pool capacity. A 4× capacity reduction minimally drops StarNUMA's average speedup from 1.54× to 1.48×. FMI is the most affected workload, with its speedup dropping from 1.22× to 1.05×. BFS and SSSP are more bandwidth-intensive than latency-sensitive compared to FMI (Fig. 8b). Thus, with a smaller pool, they benefit from leveraging both the pool's extra bandwidth, and the inter-socket links (which remain underutilized when the pool is larger). Overall, *most workloads are rather insensitive to the pool size,* indicating that a high fraction of remote accesses targets a small fraction of pages, the hottest of which still fit in the pool.

F. Page Replication versus Memory Pooling

An alternative approach to ameliorate costly remote accesses to vagabond pages is to replicate them across sockets. Page replication introduces the major overhead of maintaining coherence across page replicas in software. Besides the added software complexity, heavy read-write activity on shared pages can incur prohibitive performance overhead. Indeed, we observed high frequency of coherence transactions handled by the memory pool directory in our workloads (see §V-A). Furthermore, read-only shared pages are good replication candidates only when memory capacity waste is not a concern.

Fig. 2 in §II-B showed the page sharing degree and access distributions for BFS. BFS is representative of workloads where most accesses target shared read-write pages, resulting in a coherence action handled by the pool's directory every 50 cycles on average. In such cases, replication coupled with software coherence would incur prohibitive overheads.

Fig. 13 shows the same distributions for one more workload, TC, as it represents an entirely different workload behavior class where most memory accesses are to shared read-only pages. However, 60%/80% of the dataset is touched by 16/8+ sockets, respectively, therefore, while coherence-free, replication can cause excessive memory capacity pressure. The remaining workloads fall in between BFS and TC in page access behavior.

In summary, replication can be effective for read-only vagabond pages, when these pages *also* account for a high fraction of accesses *and* a low fraction of the memory footprint. Ultimately, page replication and StarNUMA can be jointly leveraged as complementary techniques.

G. Impact of Evaluation Methodology

Due to the system size and runtime our evaluation must cover, we devised a novel simulation methodology founded on prior best practices. To reinforce our results' robustness, we repeat §V-A's experiments for a subset of our workloads with two additional Simulation Configurations (SC), to compare against §IV's configuration **SC1** used throughput §V). Thus, we compare three simulation configurations. The default **[SC1]:** 100M instructions simulated per core every 1B-instruction phase; **[SC2]:** simulating 3× more detailed instructions (300M per 1B-instruction phase); and **[SC3]:** doubling system scale (8 cores per socket, 2× memory/interconnect bandwidth, and new traces for the doubled core count: $8 \times 16 = 128$ threads).

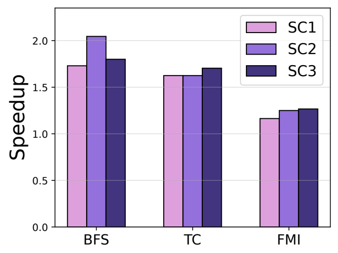


Fig. 14: StarNUMA speedup over baseline, using alternative simulation configurations.

Fig. 14 shows that while results are, unsurprisingly, not quantitatively identical, they are very close, and qualitatively all in agreement. For TC, SC2 speedup remains identical to SC1, and SC3 speedup is 4% higher. FMI's speedup also remains consistent, with SC2 and SC3 within 5% of SC1. BFS's speedup further improves over SC1's 1.7×, to 2× and 1.8× respectively. Overall, even larger and costlier simulation configurations, SC2 and SC3, confirm StarNUMA's potential, yielding similar or better results.

VI. Related Work

NUMA systems and data placement. NUMA systems date back to the early 90s, which witnessed heavy research activity around the design of large multiprocessor Distributed Shared Memory (DSM) systems. DSMs aimed to enable fast access to large physically distributed but logically shared memory, and are coarsely classified into software- and hardware-based. Software DSMs typically rely on the OS or other runtime system support to "fault in" remote memory accesses, at page [9], [15], [39] or finer [54], [55] granularity. Software's responsibility to maintain coherence by propagating data updates incurs high performance overheads, even when employing relaxed memory models. Modern software DSMs over fast RDMA networks [25], [28], [62] alleviate data movement overheads, but still encounter software bottlenecks and exacerbate programming complexity.

Hardware DSMs, or cache-coherent NUMA (ccNUMA) systems, employ hardware to enable fully transparent cache-coherent physical memory sharing. Modern multi-socket shared memory architectures descend from ccNUMA, with prominent representatives from both academia [5], [35], [37], [43], [51] and industry [49]. While hardware transparently identifies the location of each accessed cache block and performs the necessary data movements, frequent remote memory accesses degrade performance. Hence, data placement, which is typically governed at page granularity in software by the OS, plays a critical role in a ccNUMA system's performance.

Due to its impact on performance, page placement has been heavily investigated with the advent of ccNUMA systems and is still an active research topic for multi-socket machines, as well as heterogeneous and tiered memory systems comprising multiple different memories with various characteristics and constraints. Early ccNUMA work demonstrated that judicious data placement and migration can dramatically impact performance [13], [14], [36], [45], [63]. A large body of recent work focuses on software mechanisms that optimize data placement and movement in NUMA and/or tiered memory systems of various characteristics [6], [21], [34], [46], [48], [58]. Doudali et al. highlight the importance of selected page migration frequency in hybrid memory systems [22] and propose a mechanism that leverages machine learning to dynamically tune it [23]. AutoTiering [34] highlights the significance of tuning page placement and migration mechanisms to the precise characteristics of the memory system they operate on.

An orthogonal technique to increase local memory accesses and mitigate NUMA effects is selective data replication, such as replication of read-only pages [63]. R-NUMA's hardware FSM dynamically identifies opportunities for selective replication of individual cache blocks, even within read-write pages [26]. Mitosis [4] and NrOS [12] offer targeted NUMA-effect mitigations by extending the OS to selectively replicate page tables and kernel state across sockets, while Dvé [50] proposes hardware that performs cross-socket replication to improve performance and resilience. We discussed drawbacks of replication in §V-F.

CXL-based memory systems. StarNUMA's key contribution is the introduction of a new building block for NUMA systems, to facilitate placement of vagabond pages for long-latency NUMA latency mitigation. While not fundamentally tied to CXL, we argue for a CXL-based implementation due to the technology's versatility and open standard that is gaining widespread industry adoption. As CXL has the potential of being transformative in many dimensions of memory system design, the technology has garnered significant research attention. Sun et al. characterize first-generation CXL memory devices and propose guidelines for effective use in future systems [58]. Cho et al. leverage CXL's bandwidth superiority over DDR to redesign the memory system of high-throughput servers [16]. DirectCXL is one of the first system prototypes enabling CXL-based memory disaggregation, demonstrating superiority over prior equivalent RDMA-based solutions [27]. Both DirectCXL and Pond [38] demonstrate use cases of sharing a pool of CXL-attached memory across multiple hosts. Both systems focus on scale-out architectures and flexible partitioning, rather than active sharing of the memory pool across hosts. In contrast, StarNUMA demonstrates the utility of a memory pool in the context of a scale-up architecture, where all memory is actively shared by all sockets.

VII. Conclusion

In this paper, we identified that workloads with irregular memory access patterns pose a challenge in large multi-socket systems, as they exhibit a large fraction of vagabond pages—i.e., pages without a natural home socket. As a result, even intelligent data placement and migration techniques cannot eliminate costly remote memory accesses, which inflate AMAT and hurt performance. To alleviate this problem, we introduce StarNUMA, a NUMA architecture augmented with the new architectural block of a memory pool that is directly accessible from every socket. Such a memory pool, accessible 2× faster than the worst-case NUMA latency, can be implemented by leveraging the emerging CXL interconnect technology, making StarNUMA a practical design. By placing vagabond pages in the memory pool, StarNUMA significantly reduces the fraction of long-latency 2-hop memory accesses, reducing the AMAT of graph, HPC, data serving, and transactional workloads by 48% and yielding performance gains of up to 2.17×, and 1.54× on average over a typical 16-socket system.

Acknowledgments

We thank the anonymous reviewers, Byung (Michael) Choi, Marina Vemmou, Hamed Seyedroudbari, Divya Kadiyala, Pulkit Gupta, Anirudh Jain, and Anirudh Sarma for their feedback that helped improve this paper. Anand Naik contributed page access statistic studies that informed the implemented migration mechanism. This work was supported by NSF grant 2333049 and a research gift by Samsung.

References

[1] "ChampSim." [Online]. Available: https://github.com/ChampSim/ChampSim

[2] "PCI Express 6.0 Specification," 2024. [Online]. Available: https://pcisig.com/pci-express-6.0-specification

[3] S. Aananthakrishnan, S. Abedin, V. Cavé, F. Checconi, K. D. Bois, S. Eyerman, J. B. Fryman, W. Heirman, J. Howard, I. Hur, S. Jain, M. M. Landowski, K. Ma, J. A. Nelson, R. Pawlowski, F. Petrini, S. Szkoda, S. Tayal, J. J. Tithi, and Y. Vandriessche, "The Intel Programmable and Integrated Unified Memory Architecture Graph Analytics Processor," *IEEE Micro*, vol. 43, no. 5, pp. 78–87, 2023.

[4] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, "Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, 2020, pp. 283–300.

[5] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. A. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, 1995, pp. 2–13.

[6] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent Page Management for Two-tiered Main Memory," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*, 2017, pp. 631–644.

[7] AMD, "4th Gen AMD EPYC processor architecture." [Online]. Available: https://www.amd.com/system/files/documents/4th-gen-epyc-processor-architecture-white-paper.pdf

[8] N. Amit, A. Tai, and M. Wei, "Don't shoot down TLB shootdowns!" in *Proceedings of the 2020 EuroSys Conference*, 2020, pp. 35:1–35:14.

[9] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer*, 1996.

[10] T. Bang, N. May, I. Petrov, and C. Binnig, "The full story of 1000 cores," *VLDB J.*, vol. 31, no. 6, pp. 1185–1213, 2022.

[11] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP Benchmark Suite," *CoRR*, vol. abs/1508.03619, 2015.

[12] A. Bhardwaj, C. Kulkarni, R. Achermann, I. Calciu, S. Kashyap, R. Stutsman, A. Tai, and G. Zellweger, "NrOS: Effective Replication and Sharing in an Operating System," in *Proceedings of the 15th Symposium on Operating System Design and Implementation (OSDI)*, 2021, pp. 295–312.

[13] D. L. Black, A. Gupta, and W.-D. Weber, "Competitive management of distributed shared memory," in *Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage*, 1989, pp. 184–190.

[14] M. Bull and C. Johnson, "Data distribution, migration and replication on a cc-numa architecture," 2002.

[15] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991, pp. 152–164.

[16] A. Cho, A. Saxena, M. Qureshi, and A. Daglis, "COAXIAL: A CXL-Centric Memory System for Scalable Servers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2024.

[17] E. Chobanyan, C. Morrison, and P. Alavi, "End-to-end system level simulations with retimers for pcie gen5 & cxl," *DesignCon*, 2020. [Online]. Available: https://www.asteralabs.com/wp-content/themes/astera-labs/images/retimer-cxl.pdf

[18] CXL Consortium, "Compute Express Link (CXL) Specification, Revision 3.0, Version 1.0," 2022. [Online]. Available: https://www.computeexpresslink.org/_files/ugd/0c1418_1798ce97c1e6438fba818d760905e43a.pdf

[19] CXL Consortium, "Compute Express Link Consortium, Inc. and CCIX Consortium, Inc. announce agreement for CXL Consortium to receive CCIX Consortium Specifications and other CCIX Consortium assets," 2023. [Online]. Available: https://www.computeexpresslink.org/post/compute-express-link-consortium-inc-and-ccix-consortium-inc-announce-agreement-for-cxl

[20] D. Das Sharma, "The PCIe® 6.0 Specification Webinar Q&A: Supported Features in PCIe 6.0 Specification," 2024, accessed: 2024-02-22. [Online]. Available: https://pcisig.com/pcie%C2%AE-60-specification-webinar-qa-supported-features-pcie-60-specification

[21] M. Dashti, A. Fedorova, J. R. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth, "Traffic management: a holistic approach to memory placement on NUMA systems," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*, 2013, pp. 381–394.

[22] T. D. Doudali, D. Zahka, and A. Gavrilovska, "The Case for Optimizing the Frequency of Periodic Data Movements over Hybrid Memory Systems," in *Proceedings of the 2020 International Symposium on Memory Systems (MEMSYS)*, 2020, pp. 137–143.

[23] T. D. Doudali, D. Zahka, and A. Gavrilovska, "Cori: Dancing to the Right Beat of Periodic Data Movements over Hybrid Memory Systems," in *Proceedings of the 35th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2021, pp. 350–359.

[24] EE Times, "CXL will absorb Gen-Z," 2021. [Online]. Available: https://www.eetimes.com/cxl-will-absorb-gen-z/

[25] W. Endo, S. Sato, and K. Taura, "MENPS: A Decentralized Distributed Shared Memory Exploiting RDMA," in *Fourth IEEE/ACM Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*, 2020, pp. 9–16.

[26] B. Falsafi and D. A. Wood, "Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA," in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, 1997, pp. 229–240.

[27] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct Access, High-Performance Memory Disaggregation with DirectCXL," in *Proceedings of the 2022 USENIX Annual Technical Conference (ATC)*, 2022, pp. 287–294.

[28] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient Memory Disaggregation with Infiniswap," in *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 649–667.

[29] Hewlett Packard Enterprise, "HPE Superdome Flex Servers," 2023, accessed: 2023-08-10. [Online]. Available: https://www.hpe.com/us/en/servers/superdome.html

[30] IBM, *IBM Power E1080 Data Sheet*, 2021, accessed: 2023-08-10. [Online]. Available: https://www.ibm.com/downloads/cas/MMOYB4YL

[31] C. Imes, S. Hofmeyr, D. I. D. Kang, and J. P. Walters, "A case study and characterization of a many-socket, multi-tier numa hpc platform," in *IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, 2020.

[32] J. Jaffari, A. Ansari, and R. Beraha, "Systems and methods for a hybrid parallel-serial memory access," 2015, US Patent 9747038B2.

[33] Kaon Interactive, "HPE Superdome Flex Interactive," 2023, accessed: 2023-08-10. [Online]. Available: https://apps.kaonadn.net/5185710160084992/product.html#1/199;C187

[34] J. Kim, W. Choe, and J. Ahn, "Exploring the Design Space of Page Management for Multi-Tiered Memory Systems," in *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, 2021, pp. 715–728.

[35] J. Kuskin, D. Ofelt, M. A. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy, "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994, pp. 302–313.

[36] R. P. LaRowe, M. A. Holliday, and C. S. Ellis, "An Analysis of Dynamic Page Replacement on a NUMA Multiprocessor," in *Proceedings of the 1992 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1992, pp. 23–34.

[37] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford Dash Multiprocessor," *Computer*, vol. 25, no. 3, pp. 63–79, 1992.

[38] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms," in *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*, 2023, pp. 574–587.

[39] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, 1989.

[40] Linux, "Automatic NUMA Balancing," 2014, https://www.linux-kvm.org/images/7/75/01x07b-NumaAutobalancing.pdf.

[41] G. Loh, N. Jayasena, J. Chung, S. Reinhardt, M. O'Connor, and K. McGrath, "Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems," 2012.

[42] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, p. 190–200, jun 2005.

[43] K. Mackenzie, J. Kubiatowicz, A. Agarwal, and M. F. Kaashoek, "Fugu: Implementing translation and protection in a multiuser, multimodel multiprocessor," in *1994 Workshop on Shared Memory Multiprocessors*, USA, 1994.

[44] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 2012 EuroSys Conference*, 2012, pp. 183–196.

[45] M. Marchetti, L. I. Kontothanassis, R. Bianchini, and M. L. Scott, "Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems," in *Proceedings of 9th International Parallel Processing Symposium*, 1995, pp. 480–485.

[46] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. O. Kanaujia, and P. Chauhan, "TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory," in *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*, 2023, pp. 742–755.

[47] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 126–136.

[48] Y. Moon, W. Doh, K. Kyung, E. Lee, and J. H. Ahn, "ADT: Aggressive Demotion and Promotion for Tiered Memory," *IEEE Comput. Archit. Lett.*, vol. 22, no. 1, pp. 21–24, 2023.

[49] L. Noordergraaf and R. van der Pas, "Performance Experiences on Sun's WildFire Prototype," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC)*, 1999, p. 38.

[50] A. Patil, V. Nagarajan, R. Balasubramonian, and N. Oswald, "Dvé: Improving DRAM Reliability and Performance On-Demand via Coherent Replication," in *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, 2021, pp. 526–539.

[51] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-Level Shared Memory," in *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994, pp. 325–336.

[52] Riki Otaki, "tpcc-runner," 2023, accessed: 2023-08-10. [Online]. Available: https://github.com/wattlebirdaz/tpcc-runner

[53] R. Rooney and N. Koyle, "Micron® DDR5 SDRAM: New Features," *Micron Technology Inc., Tech. Rep*, 2019.

[54] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996, pp. 174–185.

[55] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain Access Control for Distributed Shared Memory," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994, pp. 297–306.

[56] D. D. Sharma, "Compute Express Link®: An open industry-standard interconnect enabling heterogeneous data-centric computing," in *Proceedings of the 2022 Annual Symposium on High-Performance Interconnects*, 2022, pp. 5–12.

[57] A. Subramaniyan, Y. Gu, T. Dunn, S. Paul, M. Vasimuddin, S. Misra, D. T. Blaauw, S. Narayanasamy, and R. Das, "GenomicsBench: A Benchmark Suite for Genomics," in *Proceedings of the 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 1–12.

[58] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, I. Jeong, R. Wang, and N. S. Kim, "Demystifying CXL memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[59] The Next Platform, "Big Iron Will Always Drive Big Spending," 2021. [Online]. Available: https://www.nextplatform.com/2021/09/21/big-iron-will-always-drive-big-spending/

[60] The Register, "CXL absorbs OpenCAPI on the road to interconnect dominance," 2022. [Online]. Available: https://www.theregister.com/2022/08/02/cxl_absorbs_opencapi/

[61] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 18–32.

[62] R. Veldema and M. Philippsen, "Evaluation of RDMA Opportunities in an Object-Oriented DSM," in *20th International Workshop on Languages and Compilers for Parallel Computing*, 2007, pp. 217–231.

[63] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996, pp. 279–289.

[64] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramírez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory," in *Proceedings of the 20th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2011, pp. 340–349.

[65] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, 2003, pp. 84–95.