THE UNIVERSITY OF CHICAGO

UDMEM: SCALABLE MEMORY MANAGEMENT FOR MASSIVELY PARALLEL DISTRIBUTED MEMORY ARCHITECTURE

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
YUQING WANG

CHICAGO, ILLINOIS
DECEMBER 2026

© 2026 by Yuqing Wang

ORCID iD: https://orcid.org/0009-0006-3548-5652

TABLE OF CONTENTS

LIST OF FIGURES vi

LIST OF TABLES x

ABSTRACT xi

ACKNOWLEDGMENTS xiii

1 INTRODUCTION 1
1.1 Motivation: Massive-Scale Global Memory Systems and the Memory Wall . . 1
1.2 Challenge 1 3
1.3 Challenge 2 5
1.3.1 Shared-State and Centralized Coordination 6
1.3.2 Dynamic Load Balancing 6
1.4 Scalable Translation and Memory Management 6
1.5 Contributions 8
1.6 Dissertation Organization 10

2 BACKGROUND 12
2.1 Virtual Memory and Address Translation 12
2.1.1 Page-based translation 13
2.1.2 Segment and range translation 16
2.2 Parallel Applications 16
2.2.1 Programming models and data distribution 17
2.2.2 Irregular application demands 18
2.3 Global Memory Architecture 19

3 RESEARCH PROBLEM AND APPROACH 22
3.1 The Core Question 22
3.2 Address Translation for Global Distributed Memory 23
3.2.1 Problem Definition 23
3.2.2 Approach 27
3.3 Scalable Memory Management 28
3.3.1 Problem Statement 29
3.3.2 Approach 29
3.4 Summary 30

4 UDTRANS: EFFICIENT ADDRESS TRANSLATION FOR DISTRIBUTED GLOBAL MEMORY 32
4.1 UDTrans Design 32
4.1.1 The 2-Dimensional Global Physical Address Space 32
4.1.2 Translation Descriptors: The Design Space and Reversibility 33

4.1.3 The Swizzle Descriptor . . . 35
4.1.4 Translation Entry and Translation Table . . . 37
4.1.5 Translation Hardware . . . 40
4.2 Methodology . . . 41
4.2.1 Simulator and Metrics . . . 41
4.2.2 Memory Trace Collection . . . 42
4.2.3 Configurations . . . 43
4.3 Evaluation . . . 46
4.3.1 Performance on CPU/GPU Traces . . . 47
4.3.2 Performance on UpDown Traces . . . 51
4.3.3 Locality Study: The Cost of Distributing Data . . . 56
4.4 The Trade-off Between Translation State and Placement Flexibility . . . 60
4.5 Summary . . . 61

5 GLMALLOC: HIERARCHICAL MEMORY MANAGEMENT FOR DISTRIBUTED MEMORY ARCHITECTURES . . . 62
5.1 Design Principles . . . 63
5.1.1 Level-One: Privileged Global Allocation . . . 64
5.1.2 Level-Two: Unprivileged, Decentralized Fine-Grained Allocation . . . 69
5.1.3 Level-Two: Hierarchical Load Balancing . . . 72
5.2 Methodology . . . 76
5.2.1 Simulation Environment and Machine Model . . . 76
5.2.2 Evaluation Metrics . . . 76
5.2.3 Workloads and Evaluation Configurations . . . 77
5.3 Evaluation . . . 80
5.3.1 GLMalloc's Scalable Two-Level Management . . . 80
5.3.2 GLMalloc Level-Two's Decentralized, Skew-Resilient Hierarchical Load-Balancing . . . 87
5.3.3 Real-World Application Performance: Scalability, Flexibility and Efficiency . . . 101
5.3.4 Comparison to Existing Allocators . . . 108
5.4 Summary . . . 111

6 DISCUSSION AND RELATED WORK . . . 113
6.1 Discussion: Implications for Future Global Memory Architecture, System Software, and Runtime . . . 113
6.1.1 Implications for Global Memory Architecture and Hardware Co-Design 113
6.1.2 Implications for Operating Systems and Memory Management . . . 115
6.2 Related Work . . . 116
6.2.1 Address Translation: Page Versus Segment Paradigms . . . 116
6.2.2 Hardware Address Distribution in Classic Distributed-Memory Machines 118
6.2.3 RDMA, CXL, and Disaggregated Memory Systems . . . 118
6.2.4 Scalable Memory Management . . . 120
6.2.5 Accelerator and Domain-Specific Global Memory Systems . . . 123

7 SUMMARY AND FUTURE DIRECTIONS 128
7.1 Insights for Global Memory Systems 128
7.1.1 Decoupling Translation Reach Granularity from Data Placement . . . 128
7.1.2 Decoupling Privileged Protection from Unprivileged Execution 129
7.2 Open Problems and Future Research Directions 130
7.2.1 Dynamic Segment Remapping 130
7.2.2 Scratchpad-Resident Multi-Class Geometric Size Bins 131
7.2.3 Multi-Dimensional and Non-Power-of-Two Swizzle Descriptors 131
7.2.4 Dynamic Interleave Adaptation 132
7.3 Conclusion 132

REFERENCES 134

LIST OF FIGURES

1.1 Scaling of Large Language Models (LLMs): model footprints grow exponentially over time, readily exceeding single-node memory capacity [80]. 2
1.2 The Global Address Translation and Memory Management Problem: applications accessing distributed global memory need to maintain hardware protection and sharing isolation while controlling physical placement for locality. 7

2.1 Intel's 5-level page table: each translation miss needs up to 5 additional memory accesses to the in-memory radix tree to resolve. 14
2.2 Size of page tables (in GB) required to support growing application data demands. 15
2.3 Data Distribution Patterns in Parallel Systems (N = 18 elements, P = 3 processors) 18
2.4 UpDown architecture (single node). 21

3.1 Memory Sharing over a 2-Dimensional Space. 24
3.2 Address Translation Problem: Given an address `va` in `VA`, translate it into a unique address `pa` in `PA`. The physical address space consists of the memory of many distributed memory nodes, forming holes in the address space. 25
3.3 Basic Machine Model: Compute elements with word-level direct access to a global memory of petabytes large. All memory accesses, both to local and remote DRAM, are controlled by translation, a low-latency network (<1 us), with high bandwidth (PB/s) connects the machine. 26

4.1 Physical address format (top) and the 2-dimensional physical address space (bottom). 33
4.2 The Design Space of Translation Schemes for Distributed Global Memory 34
4.3 Swizzle mask (top) and the corresponding address-swizzle operation (bottom). . 36
4.4 Each translation entry expresses a swizzle mapping between the virtual and physical address spaces. Contiguous address blocks in the virtual address space are mapped cyclically to blocks distributed across memory nodes. 37
4.5 A UDTrans translation entry featuring the swizzle mask, alongside the hardware logic that executes the translation. 38
4.6 UpDown Translation Table: binary lookup by the virtual addresses 39
4.7 The UD-TLB: a compact, fully associative translation cache with single-cycle access latency, shared by four compute lanes. 41
4.8 TLB hit rate across the CPU/GPU traces (per-framework arithmetic mean). Per-node segmentation is undefined (n/a) on single-node traces. 48
4.9 DRAM traffic relative to 4 KB pages on the CPU/GPU traces (each bar is a per-framework geometric mean of the total-traffic reduction ratio). 49
4.10 Speedup relative to 4 KB pages on the CPU/GPU traces (per-framework geometric mean of the ratio; higher is better). 50
4.11 Translation power relative to 4 KB pages on the CPU/GPU traces: CACTI 22 nm array power plus page-walk DRAM traffic for a single translation unit (per-framework geometric mean of the ratio to 4 KB pages; higher is better), under an assumed 2 ns per-reference time base. 51

4.12 TLB hit rate on the UpDown traces. UDTrans hit rates exceeding 99.99% represent lower bounds. 52
4.13 DRAM traffic relative to 4 KB pages on the UpDown corpus (ratio of total traffic; higher is better; the 4 KB baseline is the dashed 1.0× line). 53
4.14 Throughput-bound speedup relative to 4 KB pages on the UpDown traces (ratio; higher is better). This ratio is analytically identical to Figure 4.13, since runtime overhead equals memory amplification by construction. 54
4.15 Translation power relative to 4 KB pages on the UpDown traces: CACTI 22 nm array power plus DRAM walk traffic, aggregated across all accelerator translation units over measured execution cycles (ratio to 4 KB pages; higher is better). . . 55
4.16 Number of descriptors required to express a N-way distribution, computed from application allocation declarations. Per-node segmentation scales as $\mathcal{O}(N \cdot D)$, exceeding TLB (8 entries) and descriptor-cache (128 entries) capacities. Unmasked fine-grained segmentation (dashed) requires 1.05 to 1.95 million descriptors, whereas UDTrans maintains constant $\mathcal{O}(D)$ state complexity. 57
4.17 Overhead of $\mathcal{O}(N \cdot D)$ descriptors across four applications collected natively at each node count, SpMV spanning $N = 2$ to $N = 128$. Left: Segment TLB hit rate versus N (annotations indicate installed descriptors). Right: DRAM accesses per translation; the descriptor cache absorbs misses at low N on every workload and spills as the working set grows, though the crossing point depends on how many descriptors are concurrently live rather than on $N \cdot D$ alone. 58
4.18 Load balance versus node count on the shared SpMV trace prefix. Left: Monolithic single-descriptor allocations cannot span nodes, yielding an imbalance factor of N. Right: Per-node segmentation and UDTrans achieve balanced distributions, but with contrasting descriptor requirements: 448 descriptors versus 7 descriptors at $N = 64$. 59

5.1 The two-level memory management design and the translation path upon a memory access. 64
5.2 The two-level memory management scheme of GLMalloc. Privileged Level-One allocation occurs once per major data structure and broadcasts translation descriptors to accelerator devices. Subsequent fine-grained allocations are served by unprivileged distributed Level-Two. 65
5.3 `DRAMalloc` 2-dimensional allocation with R-tree. 66
5.4 One parameter decides whether the lower level is local or remote. 68
5.5 Level-Two metadata caching hierarchy across the two allocator backends. . . . 70
5.6 Three-tier hierarchical load balancing in GLMalloc. 73
5.7 Impact of swizzle translation parameterization on cross-node memory traffic and application execution time across four distributed workloads. 81
5.8 Level-One segment allocation latency across machine size. Allocation cost remains flat as the machine scale. 83
5.9 Allocation and deallocation latency of a single outstanding request across both allocator backends and residency policies and physical placements, compared against a local DRAM word round trip (400 cycles dash line). 86

5.10 Cost of one allocation at each tier of the allocation path (top: slab backend, bottom: buddy backend). 87
5.11 Allocation latency under various sub-pool occupancies(top: slab backend, bottom: buddy backend). When load balancing is disabled, overflow requests fail unconditionally (red open bullets). 88
5.12 Tier 3 inter-node request forwarding. **Left**: delivered requests with respect to demand. With Tier 3 (filled), the requested memory that can be served equals the global capacity. **Right**: the price of Tier 3: Each additional forwarding hop costs cycles, from 1.15× with 1 hop to 65× with 3.9 hops. 91
5.13 Segment layout and Tier 3 rerouting. **Left**: execution cycles for every layout, normalized to node-local. **Right**: skewed applications are latency bound. . . . 92
5.14 Delivered capacity and execution cost under intra-device load skew. Rotated rerouting achieves 100% capacity utilization and reduces cost per successful allocation by 29–36%. 93
5.15 Load-balancing capacity and execution cost under machine scaling from 1 to 32 nodes (2,048 to 65,536 lanes) with per-device demand and per-node demand skew. 94
5.16 The comparison between centralized versus decentralized rerouting under intra-device skew ($H \in [1, 32]$ hot lanes of 64 per device, each demanding 200% of local pool capacity). 95
5.17 Capacity conversion under inter-device load skew (U of the node each receive 200% of their own capacity). 97
5.18 Inter-device request migration recovering full delivered capacity on the `frontier_msr` pipeline under device oversubscription. 98
5.19 Execution time and allocation success rate on `bfs_dynarray` across 1, 2, and 8 nodes. 98
5.20 The skew threshold on `graph_ingest` (16,384 lanes). The busiest lane crosses private capacity at $\theta = 0.86$, where $\sigma_{\text{lane}} > 1/\rho = 4.362$. Rerouting absorbs all overflow, and execution cost remains flat as capacity grows. 99
5.21 Failure rate under dynamic allocate-and-free as a function of retention. 100
5.22 Weak-scaling throughput and per-lane execution cost for `llm_serve` from 2 to 64 nodes (4,096 to 131,072 lanes). 101
5.23 Strong-scaling throughput and per-lane execution cost for `llm_serve` from 2 to 64 nodes (4,096 to 131,072 lanes). 102
5.24 The cost of concentrating requests. Concentrating 32 requesters onto a single allocator lane reduces aggregate device throughput by 42.4× and surges request queues by 39×. 103
5.25 Aggregate throughput under a single shared allocation structure (contended manager) versus private per-lane sub-pooling across allocator lane counts $A \in [1, 32]$. 104
5.26 Allocation latency across a 16× block size granularity sweep on `llm_serve`, varying by only 12.8% between 256 B and 4 KB. 104
5.27 System throughput as a function of pool occupancy on `llm_serve`, identifying the operational admission knee at 82% occupancy. 104

5.28 End-to-end token generation latency on `llm_serve` as a function of block size. Sizing blocks to 512 B achieves 22% faster token processing than 4 KB blocks by eliminating zero-padded KV cache memory traffic. 105
5.29 Internal memory fragmentation in KV cache management across block sizes as a function of sequence length, dropping from 11.81% at 4 KB to 0.00% at 256 B. . 105
5.30 Physical memory footprint on scale-26 BFS traversal. Static worst-case reservation exceeds the 64 GB budget (112.9%) and aborts, whereas GLMalloc completes within 31.4% of available memory. 106
5.31 Allocator backend trade-off on `graph_ingest` 107
5.32 Microarchitecturally invariant execution costs across memory allocators on an unloaded fast path. GLMalloc executes in 73 instructions and 18 private scratchpad references with zero DRAM accesses and zero atomic operations, compared to 31.7–127 references for production CPU host allocators. 108
5.33 Delivered capacity under spatial demand skew across allocators. 109
5.34 Weighted execution cost under spatial demand skew. 111

6.1 Comparison between GPU Supercomputer, TPU SuperPod, and UDMem global memory machine 125

LIST OF TABLES

4.1 Workload traces collected from CPU/GPU. Total trace memory accesses (N) range from 0.07 G to 22.44 G operations across workloads. All CPU traces use $P \in \{64, 128\}$ ranks except natively instrumented GAPBS and XSBench (P=1); GPU trace uses a single NVIDIA A40 device. Traces follow the unified allocation model (one contiguous rank share per node). 44
4.2 Application traces collected from the UpDown architecture simulator (BASim). Each application is evaluated with its memory footprint held fixed across node counts $N \in \{2, 4, 8, 16, 32, 64, 128\}$ (2,048 lanes/node), rendering trace volume invariant in N. Each data structure corresponds to one UDTrans global memory entry. 45
4.3 Simulated translation schemes. All four schemes see the identical access stream from each trace; the segment arm's per-node split uses the trace's own rank count N. 45

5.1 Evaluation workloads and dataset parameters spanning microbenchmarks and four distributed accelerator application pipelines. 78
5.2 GLMalloc evaluation configuration space across architectural axes, policies, skew models, scale, and comparative baselines. 79
5.3 Level-One segment allocation performance and two-dimensional free space fragmentation under synthetic segment-request workloads [116]. 85
5.4 GLMalloc Level-Two's metadata size across backends and configurations. 89

ABSTRACT

In recent decades, emerging high-performance big data applications have scaled by orders of magnitude in both data and computation. To overcome single-node systems' limits, modern architectures aggregate hundreds to thousands of compute and memory nodes into distributed global memory systems. As system memory scales to petabytes, memory management becomes increasingly challenging: in common distributed programming models like Message Passing Interface (MPI) and PGAS, programmers often fall back to manual data management across distinct address spaces, which compromises programmability and machine portability.

We believe global memory management and virtualization are essential for programming efficiency, software complexity, and application flexibility in distributed systems. Yet, existing solutions fall short. Traditional page translation cannot scale because the page table grows linearly as the data exceeds the TLB reach, whereas segmentation fails to exploit data locality or distributed memory bandwidth. Conventional memory management prioritizes generality. However, modern applications have drastically different memory allocation patterns, ranging from monolithic, one-time allocations to high-frequency, fine-grained requests, creating severe performance bottlenecks for global memory managers. A system-wide unified allocator cannot handle this demand variation efficiently.

To address these challenges, we propose **UDMem**, a global memory management scheme tailored for massively parallel, distributed-memory environments. UDMem consists of two components:

- **UDTrans**: A compact, scalable, segment-based address translation scheme for scale-out distributed memories. It delivers high-performance translation with negligible overhead while exposing programmer-controlled data layouts across memories for data locality.

- **GLMalloc**: A high-throughput, hierarchical memory management scheme, combining

a locality-aware, privileged global allocator with a throughput-optimized user-level allocator for highly skewed, fine-grained allocations.

We evaluate UDMem across comprehensive trace-driven and cycle-accurate simulations and show that it eliminates address-translation DRAM traffic amplification with $> 99.99\%$ TLB hit rates, delivers atomic-free fast-path allocations as long as 77 cycles while sustaining over 500 billion allocation operations per second and improves end-to-end application performance by up to $1.49\times$ via locality-aware hardware-software co-design.

In summary, UDMem provides high-throughput and energy-efficient global memory management across various applications, promoting programmability and portability for future distributed global-memory systems.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Andrew A. Chien, for his invaluable guidance, continuous support, and mentorship throughout my doctoral journey. His visionary perspective on computer systems architecture, unwavering high standards, and insightful feedback have fundamentally shaped this dissertation and my growth as a researcher.

I would also like to thank my dissertation committee members, Professor Ian Foster and Haryadi Gunawi, for their time, critical evaluation, and constructive feedback.

My doctoral experience would not have been the same without the camaraderie and collaboration of my colleagues in the Large-Scale Systems Group and the UpDown team. I am especially grateful to Andronicus Rajasukumar, Tianshuo Su, Rajat Khandelwal, Jiya Su, Ruiqi Xu, Tianchi Zhang, Jianru Ding, Wenyi Wang, Charles Colley, Alexander Fell, and Marziyeh Nourian. Working alongside such talented, supportive, and dedicated teammates made tackling complex architectural simulations and paper deadlines inspiring and rewarding.

I would also like to extend my appreciation to the faculty, staff, and students of the Department of Computer Science and the Division of the Physical Sciences at the University of Chicago for fostering an intellectually stimulating and supportive academic environment.

Finally, I am deeply grateful to my family and friends. To my parents, thank you for your unconditional love and unwavering belief in me through every step of my life. To my dearest friends, thank you for your patience, encouragement, and companionship that brought warmth and balance to my life during this journey.

CHAPTER 1

INTRODUCTION

1.1 Motivation: Massive-Scale Global Memory Systems and the Memory Wall

In the modern era of large language models (LLMs), trillion-edge graph analytics, and massive recommendation systems, the fundamental computational bottleneck has shifted from processor arithmetic throughput (the rate at which execution units could perform integer and floating-point operations) to memory. Today, large-scale systems are bounded by the Memory Capacity and Bandwidth Wall: the severe limitation in how rapidly, efficiently, and coherently data can be delivered from memory subsystems to execution units.

As illustrated in Figure 1.1, contemporary workloads exhibit exponential growth in memory footprint [80]. Modern frontier model parameters, optimizer states, and dynamic Key-Value (KV) cache activations require tens of terabytes to petabytes of memory, far exceeding the high-bandwidth memory (HBM) capacity of any single node. Accommodating these footprints necessitates scale-out architectures spanning hundreds or thousands of interconnected chips. However, distributing data across physical memory nodes introduces steep performance penalties. The physical memory hierarchy in such systems is non-uniform by design: remote memory access latencies over high-speed networks are an order of magnitude higher than local access latencies (e.g., 6.5× higher in the architecture evaluated in this work). Therefore, physical data placement and memory locality govern application performance.

To overcome the capacity wall, modern architectures are converging toward **Global Memory Systems**. These systems abstract physically distributed memory nodes into a unified, globally accessible address space. Examples include hardware-coherent GPU clusters (such as Nvidia NVLink and Grace Hopper systems) [39, 87], memory disaggregation fabrics based on Compute Express Link (CXL) [74], and near-memory, event-driven acceler-

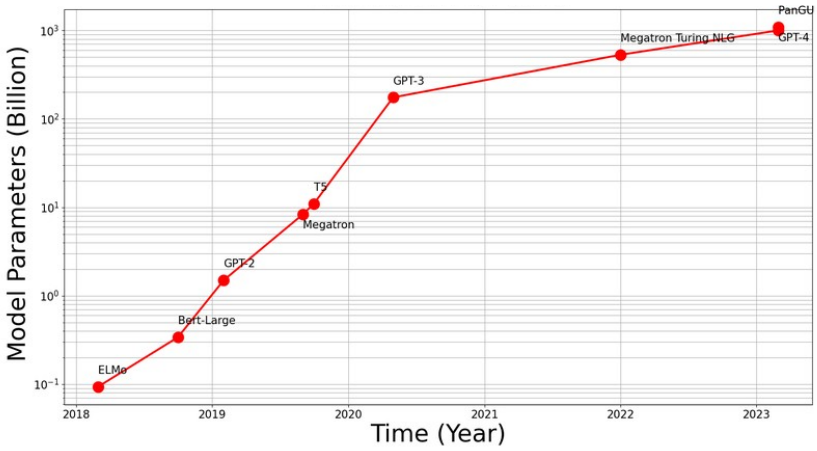


Figure 1.1: Scaling of Large Language Models (LLMs): model footprints grow exponentially over time, readily exceeding single-node memory capacity [80].

ator architectures like the UpDown supercomputing system [25, 9, 53].

While global memory systems provide the memory capacity required for petabyte-scale workloads, managing this distributed physical memory introduces a fundamental tension between programmability and performance:

1. **Address Translation Overhead**: In traditional distributed-memory systems, processes operate on independent virtual address spaces, protected and translated via page tables. However, conventional page-based translation was introduced when memory was scarce (i.e., less than megabytes) and systems were single-processor. At petabyte scale, page tables consume gigabytes of physical memory and induce massive Translation Lookaside Buffer (TLB) thrashing, spending up to 51% of execution cycles on translation [76, 13]. Conversely, conventional segmentation schemes scale with constant translation state but enforce rigid physical contiguity, preventing data structures from distributing across multiple nodes to harness aggregate memory bandwidth.

2. **Memory Management and Concurrency Collapse**: Traditional memory allocators are designed for flat, symmetric address spaces. When applied to distributed systems with millions of concurrent processing cores/lanes, centralized memory man-

agement creates severe lock contention and arbitration bottlenecks. Conversely, while fully decentralized, private memory pools eliminate synchronization on allocation, they suffer severe capacity exhaustion under non-uniform, skewed allocation demands, leaving up to 50% of available memory stranded.

To realize the full potential of next-generation distributed global memory systems, this dissertation addresses two central questions:

1. Can we design an address translation scheme that provides a protected virtual memory abstraction with minimal, constant-sized translation state while granting software fine-grained control over physical data layout?

2. Can we design a hierarchical memory management scheme that eliminates centralized management bottlenecks, guarantees processes' address space isolation, and sustains millions of concurrent, skewed, fine-grained allocations?

1.2 Challenge 1

Virtual memory is an important abstraction in modern computing. It isolates application processes from one another, eliminates manual physical memory sharing and management, and enables memory oversubscription. Address translation is the fundamental mechanism enforcing this abstraction, dynamically mapping process-virtual addresses (`VA`) to global physical addresses (`PA`) on every memory access. However, at petabyte scale, conventional address translation schemes fail to scale.

Page-Based Schemes: Massive Translation State and Limited Reach Page-based translation maps virtual pages to physical frames at a fixed granularity (typically 4 KB or 2 MB). As a result, the translation state scales linearly with the application's memory footprint ($\mathcal{O}$(pages)). While hardware TLBs cache active mappings to accelerate translation,

TLB reach remains bounded by chip area and timing constraints. When active working sets span hundreds of gigabytes, TLB reach thrashing becomes unavoidable. On irregular workloads (such as graph analytics and sparse matrix computations), 256-entry TLBs achieve hit rates below 1%, forcing nearly every memory reference to traverse a multi-level radix page table in physical memory. In a distributed topology, these page table walks traverse high-latency network fabrics, causing a geometric-mean 172% memory-access amplification (nearly tripling total DRAM traffic) and substantially degrading performance [76, 13, 57]. Huge pages (2 MB or 1 GB) extend reach proportionally but still scale linearly with footprint, demand physical contiguity that long-running systems cannot guarantee, and coarsen the placement unit, making fine-grained distributed layout impossible.

Segment-Based Schemes: High Reach but Rigid Placement Segment-based translation schemes decouple translation state from footprint size: a single translation descriptor covers an arbitrarily large contiguous memory range with $\mathcal{O}(1)$ metadata complexity [13]. However, conventional segment descriptors enforce strict contiguity in both virtual and physical address spaces. They contain no mechanism to express physical placement across discrete memory nodes. To distribute an application data structure across N nodes to exploit memory-level parallelism, conventional segmentation needs to allocate N independent descriptors per data structure, resulting in $\mathcal{O}(N \cdot D)$ translation complexity for D allocations. When scaling to dozens or hundreds of nodes ($N \geq 32$), this descriptor explosion thrashes small on-chip segment TLBs and descriptor caches, spilling translation metadata into DRAM. Furthermore, mapping fine-grained block-interleaved distributions using conventional segments requires millions of discrete descriptors, degenerating into an unmanageable page table.

The Structural Coupling of Reach and Translation Granularity The root failure of prior approaches lies in the coupling between translation and data placement granularity.

Page tables give placement flexibility at the cost of linear translation state; flat segmentation gives unbounded reach at the cost of placement rigidity. The challenge is therefore: **how to achieve unbounded translation reach with compact, non-scaling state while preserving software-guided, fine-grained data locality control across distributed physical memory nodes?**

1.3 Challenge 2

Distributed global memory cannot be viewed as a flat, monolithic space due to the gap in memory access latency and bandwidth between local and remote nodes. Managing distributed memory, therefore, requires allocators to explicitly reason about physical node topology for performance, that is, treating memory as a non-uniform, 2-dimensional address space.

Meanwhile, modern software trends have diverged from static, coarse-grained buffer allocation to massive concurrency, fine granularity, and dynamic data-dependent allocations:

- In Large Language Model serving, mechanisms such as PageAttention [64] dynamically allocate and deallocate fine-grained KV cache blocks across varying prompt lengths and concurrent generation phases.
- In distributed graph processing and sparse linear algebra, algorithms such as Breadth-First Search (BFS) and Graph Ingestion exhibit bursty, exponential frontier expansions, requiring high-rate dynamic chunk allocation.

When deployed across scale-out systems with tens of thousands to hundreds of thousands of parallel execution units , traditional memory management architectures suffer from two major failure modes:

1.3.1 Shared-State and Centralized Coordination

Traditional allocators rely on centralized metadata structures, shared free lists, or coarse-grained mutexes. When hundreds of thousands of execution units issue concurrent memory requests, shared locks and centralized arbiters become insurmountable bottlenecks. Centralized mediation introduces a 4.05× runtime penalty, while shared locks degrade runtime by 6.74× under concurrent allocation pressure.

To eliminate lock contention, high-performance allocators allocate memory from private, thread-local or lane-local arenas. However, under non-uniform or power-law workload skew (where a small subset of execution contexts performs the vast majority of allocations), private arenas suffer from the **Private-Arena Capacity Floor**: active lanes prematurely exhaust their statically partitioned local arenas while adjacent idle lanes strand substantial unallocated memory. Under severe skew, this imbalance causes premature allocation failure and aborts, stranding up to 50% of system memory capacity despite abundant global headroom.

1.3.2 Dynamic Load Balancing

Resolving data-dependent load imbalance requires allocators to dynamically monitor allocator status and adjust allocations accordingly for both performance and pool utilization/fullness. The challenge is: **how to provide a lock-free, high-throughput, low-latency fine-grained allocation while dynamically recovering load imbalance under severe skew without centralized serialization?**

1.4 Scalable Translation and Memory Management

This dissertation argues that a hardware co-designed memory management scheme, including **scalable address translation and hierarchical memory management**, can resolve global memory management challenges in massive-scale distributed memory systems reaching

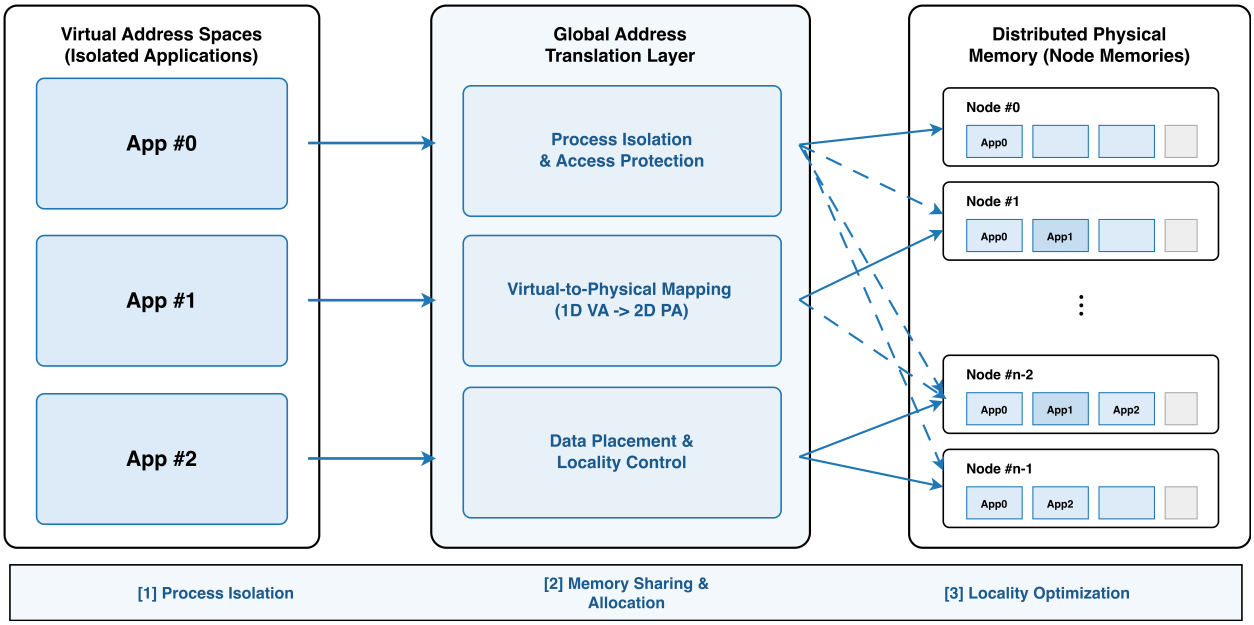


Figure 1.2: The Global Address Translation and Memory Management Problem: applications accessing distributed global memory need to maintain hardware protection and sharing isolation while controlling physical placement for locality.

petabytes.

As illustrated in Figure 1.2, hardware translation and software memory management represent two halves of a cohesive solution:

- **Scalable Address Translation (UDTrans)**: By pairing unlimited reach segment descriptors with configurable 64-bit swizzle masks (B, C), UDTrans decouples translation granularity from data placement. It bounds translation state to $(\mathcal{O}(D))$, where D is the number of global data structures, independent of data footprint or node count, while supporting 2D block-cyclic placement in hardware with single-cycle TLB access for translation.

- **Hierarchical Software Memory management scheme (GLMalloc)**: By decoupling privileged global manager from unprivileged, decentralized fine-grained allocator, GLMalloc amortizes privileged segment setup across millions of user-level allocations. The user-level allocator guarantees an atomic-free, zero-DRAM allocation fast path,

and supports a 3-tier decentralized load-balancing scheme that recovers from dynamic, data-dependent skewness.

A valid solution should satisfy the following four goals:

1. **Compact, Scalable Translation State**: Cover petabyte-scale global address spaces using $\mathcal{O}(1)$ translation state per data structure, maintaining high TLB hit rates and eliminating DRAM page walks.

2. **Software-Visible 2D Locality Control**: Expose physical node and offset dimensions directly to software, enabling applications and runtimes to eliminate remote memory traffic on parallel phases without re-implementing the program.

3. **Decoupled, Bottleneck-Free management**: Decouple privileged global physical memory partitioning from unprivileged local memory allocation, eliminating centralized locks and mediation from the allocation critical path.

4. **Skew-Resilient, High-Throughput Memory Allocation**: Sustain millions and billions of allocations per second with a deterministic, atomic-free fast path while dynamically recovering 100% of stranded memory capacity under severe workload skew.

1.5 Contributions

This dissertation makes the following contributions to address the global memory management challenge and is systematically validated through cycle-accurate simulation and extensive workload characterization:

1. **The UDTrans Swizzle Address Translation Scheme (Chapter 4)**: We design UDTrans, an address translation scheme that decouples translation granularity from data placement, and evaluate it on CPU, GPU, and accelerator traces.

- Translation State Compactness: Proved UDTrans's translation state is independent of node count N or footprint size, fitting within an 8-entry TLB and eliminating multi-level translation table walks.
- TLB Reach: Evaluated across CPU/GPU workload traces and UpDown accelerator traces, showing that UDTrans achieves $> 99.99\%$ TLB hit rates with zero DRAM access amplification, and incurs only compulsory misses across all native UpDown accelerator workloads.
- Energy and Scalability: Reduces translation power dramatically by 19×-116× (geometric mean 75×) on CPU/GPU traces and 29×-139× on UpDown traces.

2. **Compositional Two-Level Memory Management Scheme (Chapter 5)**: We propose GLMalloc, a two-level memory management scheme that decouples privileged global memory access control and data placement from unprivileged, decentralized fine-grained allocation. It provides a translation and allocation solution that enables flexible, multi-tenant use of a global memory system.
   - Segment Allocation: Prove that the cost of GLMalloc privileged allocation can be amortized across unprivileged user-level allocations and that UDMem supports multi-tenant global memory sharing at low cost.
   - Fast Path Allocation: Show that the fast path executes in as few as 77 cycles without atomic operations.
   - Locality Control: Prove that exposing hardware swizzle parameters (B, C) to runtime software eliminates cross-node memory accesses, yielding speedups between 1.02× and 1.49× across four distributed application workloads.
3. **Decentralized, Skew-Resilient Hierarchical Load Balancing (Chapter 5)**: We design an unprivileged load-balancing management scheme for user-level fine-grained allocation.

- Three-Tier management: Present the 3-tier design including intra-device rotated-origin rerouting (Tier 1), inter-device sub-pool region redistribution (Tier 2), and inter-node forwarding (Tier 3) to skewed allocation distribution.
- Capacity Recovery: Recovers the private-pool capacity floor under power-law skew ($\theta \in [0.65, 0.99]$) and hot-spot, serving all allocations while cutting average allocation latency by 29%-36% via maximizing local allocation.
- Decentralization and Sharing-free: Compares GLMalloc to centralized management and shared-locks management, showing that decentralization achieves 4.05× and 6.74× improvements in runtime.

4. **Application Performance and Scalability (Chapter 5)**: We evaluate GLMalloc across four application workloads, scaling to millions of parallel processing cores, and illustrate that GLMalloc enables flexible use of global memory.
   - Massive Parallel Scaling: Achieves 98% weak-scaling efficiency up to 131,072 processing lanes (64 nodes), sustaining an aggregate throughput of 269.88 allocations per processor cycle, exceeding 500 billion allocations/sec at 2 GHz globally.
   - Backend Flexibility: Demonstrates the benefits of backend flexibility (buddy vs. slab), reducing memory footprint by 15×-16× on skewed allocations.
   - Comparison to Existing Allocators: GLMalloc outperforms production host allocators (TCMalloc, jemalloc, and glibc malloc) by executing 1.76×–7.06× fewer memory-hierarchy references per allocation.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows:

- **Chapter 2 (Background)**: Reviews the virtual memory literature (page-based versus segment-based translation mechanisms), distributed parallel programming models

(MPI, PGAS), and scale-out global memory architectures.

- **Chapter 3 (Research Problem and Approach)**: Formulates the 2D global address translation and memory management problem, details the performance and hardware budget requirements, and presents the UDMem hardware-software co-design for global memory architecture approach.

- **Chapter 4 (UDTrans: Efficient Address Translation for Distributed Global Memory)**: Presents the design of UDTrans, illustrating the swizzle descriptors over the 2D physical address space, translation table, and hardware translation structures. Evaluates translation overheads, memory amplification, power consumption, and locality scaling across CPU/GPU and UpDown accelerator workload traces.

- **Chapter 5 (GLMalloc: Hierarchical Memory Management for Distributed Memory Architectures)**: Presents the architecture, fast-path invariants, and 3-tier skew-resilient load balancing of GLMalloc. Evaluates performance scaling up to millions-fold parallelism, characterizes capacity recovery under severe skew on real-world workloads, validates UDTrans co-design, and positions performance against production memory allocators.

- **Chapter 6 (Discussion and Related Work)**: Discusses the architecture and software runtime implications drawn from evaluating UDMem and provides a comprehensive review of prior art across page-table optimization, hardware segmentation, translation-allocator co-design, distributed memory management, and scalable concurrent allocators.

- **Chapter 7 (Summary and Future Directions)**: Concludes the dissertation by summarizing our findings and identifying promising future directions for scale-out memory management.

CHAPTER 2

BACKGROUND

In Chapter 1, we argue that realizing a unified, petabyte-scale address space across thousands of distributed nodes requires simultaneously satisfying three performance constraints: 1) address resolution should incur minimal latency overhead, 2) application processes' translation metadata should remain small and compact, and 3) parallel applications should retain flexible control over physical data placement. This chapter presents background on memory management for scalable distributed memory systems and parallel programming and formalizes the three foundational requirements.

Section 2.1 examines the conventional page- and segment- translation schemes common on single-node shared memory systems and identifies how these mechanisms couple translation reach to placement granularity and fail to deliver scalable performance in scale-out scenarios. Section 2.2 characterizes the parallel application workloads which drive this demand on expressing data layout, demonstrating how dynamic and data-dependent memory access patterns undermine traditional translation schemes. Finally, Section 2.3 outlines contemporary distributed memory architectures and defines the 2-dimensional address space model assumed throughout this work.

2.1 Virtual Memory and Address Translation

Virtual memory provides an indirection layer between the application and physical memory, decoupling an application's logical namespace from physical memory locations and enabling memory isolation and sharing. Since the introduction of the Multics virtual memory architecture [16], this abstraction layer has carried out three distinct responsibilities simultaneously: translating virtual addresses to physical locations, isolating inter-process address spaces, and orchestrating data layout across distributed physical memories.

In single-node environments, these three functions are tightly coupled within the operating-system's address translation structures. In distributed scale-out architectures, however, these responsibilities diverge, demanding an explicit re-evaluation of the guarantees provided by each architectural layer.

2.1.1 Page-based translation

Conventional virtual memory systems fix the allocation at a single granularity (4 KB or 64 KB) and maintain a multi-level hierarchical page table to map virtual pages to physical frames. The active translations are cached in Translation Lookaside Buffers (TLBs). This design imposes two distinct, orthogonal performance overheads to applications.

The first overhead is the latency of translation/Page Table walk upon a TLB miss. Serving a miss in TLB requires a multi-level radix-tree walk, each of which is an expensive memory access: as the memory grows from tera to petabytes, the depth of the Page Table increases from 4 to 5 levels, incurring an even larger penalty for TLB misses [30, 31]. Prior work mitigates the issue through several optimization strategies: caching intermediate page-walk levels via page-walk caches [11], provisioning dedicated large-capacity MMU caches [18], designing page-table layouts that resolve walks within a single cache hit [93], prefetching translations ahead of demand [76], and preloading entries based on recency heuristics [104]. While these mechanisms reduce the average latency of page-table walks, they do not alter the underlying page granularity, and the resulting page-table depth remains proportional to the virtual memory size.

The second overhead is finite TLB reach, which is determined by the product of the TLB entry count and the page size. Once the application's working set size exceeds this capacity threshold, miss rates are determined entirely by the working set's spatial locality rather than by caching efficiency. For large-memory workloads, address translation overheads consume significant fractions of execution time, up to 51% of total cycles in worst-case scenarios [13,

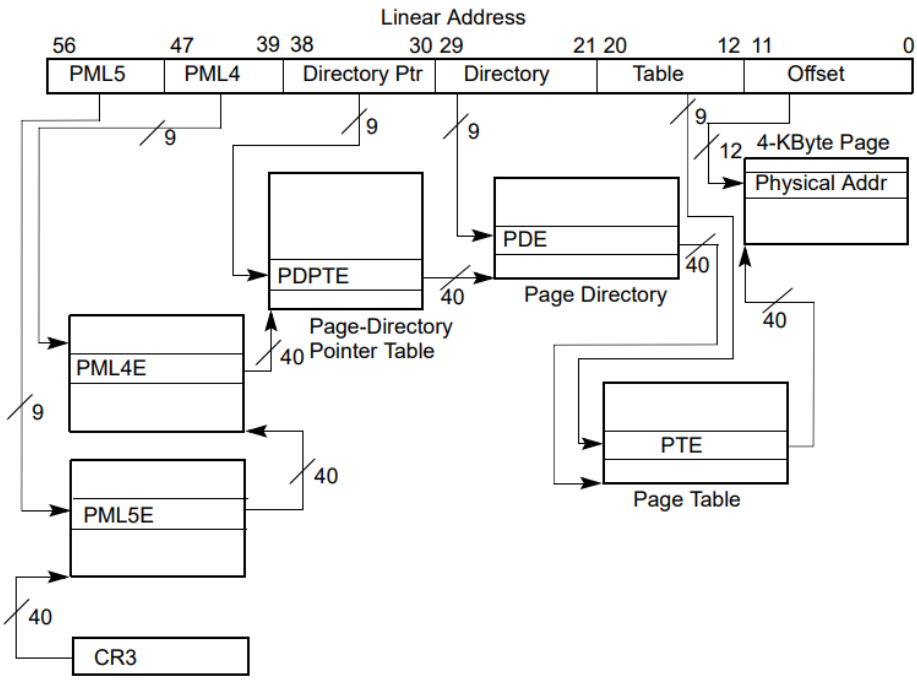


Figure 2.1: Intel's 5-level page table: each translation miss needs up to 5 additional memory accesses to the in-memory radix tree to resolve.

57]. Because the translation granularity (page size) is fixed, translation state scales proportionally to the mapped address space: each mapped byte requires a fixed proportional fraction of page-table metadata, irrespective of the aggregate allocation size. Figure 2.2 shows that the page table size is estimated to reach tens to hundreds of terabytes as physical memory scales to petabytes of memory as the data set size continues to scale.

Huge pages expand translation reach by supporting various, larger page granularities than the base translation unit [27, 28]. While huge pages scale reach and substantially reduce page-table capacity requirements, the asymptotic relationship between translation state and aggregate mapped memory remains linear for any fixed granule page-based scheme; increasing page size shifts the capacity threshold without changing the scaling behavior. Furthermore, huge pages impose strict physical contiguity constraints that degrade under memory fragmentation in long-running systems [123], while the policies governing page promotion introduce complex runtime trade-offs [65]. Software and hardware techniques that

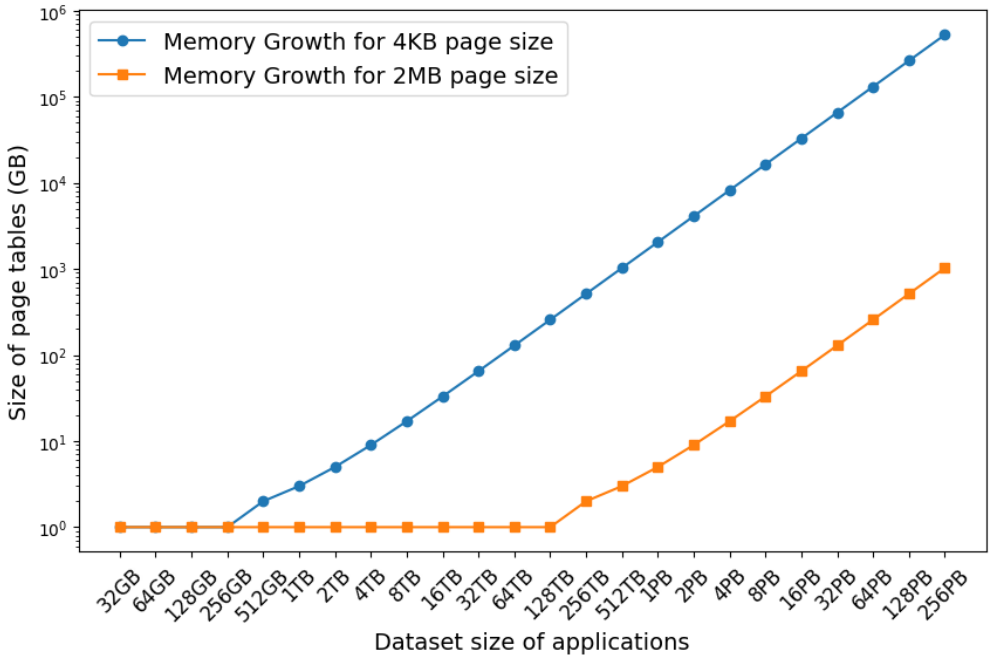


Figure 2.2: Size of page tables (in GB) required to support growing application data demands.

coalesce adjacent translations into single TLB entries [95], or sustain coalescing across page migrations in heterogeneous memory tiers [115], partially mitigate reach limitations but remain constrained by the degree of physical contiguity produced by the underlying memory allocator.

The fundamental limitation lies in the structure of page coupling reach with placement: a page serves simultaneously as the atomic unit of address translation and the atomic unit of physical placement. Therefore, expanding translation reach by increasing page size necessarily coarsens the granularity at which physical placement can be expressed. This structural tension between translation reach and placement granularity motivates the architectural design proposed in this dissertation.

2.1.2 Segment and range translation

Eliminating the fixed-granule assumption decouples translation state from total allocated capacity. A segment descriptor, which is defined by a virtual base, a physical base, and a region length, translates an arbitrarily large contiguous address range using a single entry. Therefore, translation state scales with the number of discrete allocated regions rather than the aggregate mapped memory footprint [13]. Extensions of this model support multiple variable-sized ranges per process [57], integrate range comparators alongside tagged pointers directly within the TLB [24], and have been implemented in physical hardware [63]. Alternative architectures redesign the access path to bypass traditional virtual-to-physical translation entirely [50].

However, conventional segment descriptors cannot represent data layouts distributed across physical memory. Standard segment descriptor fields encode only a contiguous physical range and lack explicit semantic identifiers to designate target nodes within a distributed topology. While page-based systems express per-node placement at page granularity through operating-system allocation policies, i.e., default first-touch allocation [67], explicit NUMA policies via `numactl`, `libnuma`, and the underlying `mbind`/`set_mempolicy` interfaces [61], dynamic page migration [46], or distributed page-table and TLB management across NUMA nodes [41], segment translation sacrifices placement granularity and flexibility to maximize translation reach. Thus, page tables optimize placement at the expense of reach, whereas segment schemes maximize reach at the expense of fine-grained placement expressiveness.

2.2 Parallel Applications

The efficiency of an address translation architecture is governed by the access patterns of the applications running on it. This section reviews distributed parallel programming models and focuses on applications whose memory access profiles are dynamically determined by runtime input data, i.e., irregular and data-intensive.

2.2.1 Programming models and data distribution

The standard paradigm for distributed-memory computing relies on isolated address spaces. Under the Message Passing Interface (MPI) standard [113], each process rank operates within a private address space, with cross-node communication orchestrated via explicit data transfers between disjoint memory domains. Therefore, distributed data structures need to be partitioned manually, cross-node references are encoded as explicit rank-and-offset tuples managed within application logic, and modifying the spatial data layout requires structural code refactoring.

Partitioned Global Address Space (PGAS) frameworks provide a unified global address space while exposing physical locality to software [7]. In these models, the programmer specifies data distribution policies that are instantiated by the runtime environment: UPC partitions shared arrays across execution threads via user-defined block factors [42], Chapel provides `Block` and `Cyclic` domain maps as core language constructs [21], OpenSHMEM standardizes symmetric memory regions accessed through one-sided communication primitives [15], Global Arrays defines array distributions at allocation time [83], X10 defines first-class locality domains via places [23], and UPC++ integrates PGAS abstractions into modern C++ library frameworks [10]. Canonical distribution strategies, such as block-cyclic decomposition [92], remain foundational for distributed linear algebra libraries [26] and represent common layout patterns required by high-performance applications. Figure 2.3 illustrates the common data distribution patterns PGAS programs use for global data.

The important point is that distribution directives in PGAS models are evaluated predominantly by software runtime layers. Placement resolutions occur asynchronously during allocation or explicit communication operations via general-purpose instructions, rather than being evaluated synchronously by hardware translation mechanisms on every load and store. The motivation for our work is to embed the distribution directly into address translation, thereby providing a simpler, more efficient framework for parallel programs.

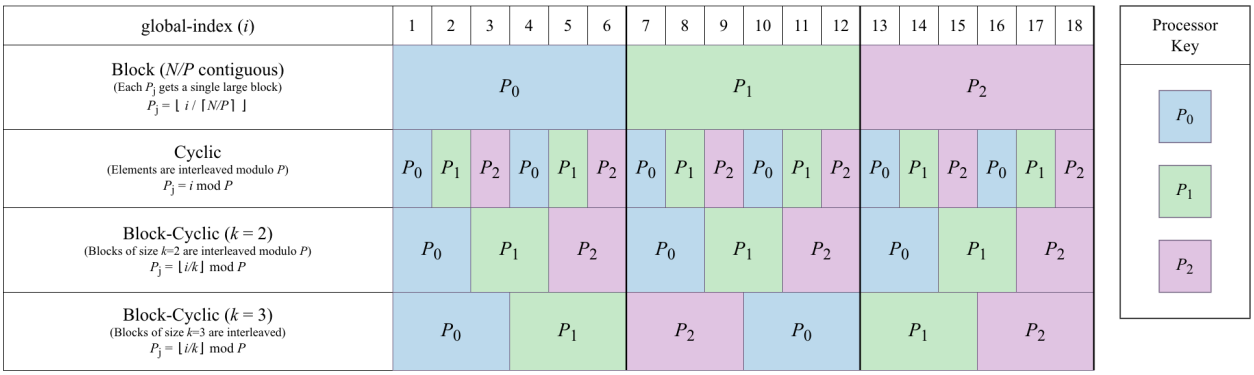


Figure 2.3: Data Distribution Patterns in Parallel Systems (N = 18 elements, P = 3 processors)

2.2.2 Irregular application demands

Structured and dense linear algebra computations feature static data access patterns that can be effectively partitioned at compile time or during initialization. In contrast, applications with dynamic access characteristics largely undermine static partitioning schemes. We use two prominent classes of modern workloads that exhibit these patterns to illustrate the challenge.

Irregular graph analytics involves traversals over topologies whose structure depends entirely on input datasets. In algorithms such as breadth-first search (BFS), the set of active frontier vertices at a given iteration (and the corresponding memory footprint and access locality) cannot be predicted prior to runtime execution. Standardized benchmark suites [48, 49] and generative graph models [33] generate sparse, high-entropy memory access streams spanning massive virtual address spaces, degrading the hit rates of capacity-constrained TLBs. Similar access characteristics arise in continuous-energy Monte Carlo particle transport simulations [120].

Large language model inference presents comparable dynamic memory demands within high-throughput serving pipelines. Token sequence lengths vary non-deterministically across requests, causing Key-Value (KV) cache allocations to expand and contract dynamically. Allocating contiguous memory blocks based on worst-case sequence lengths intro-

duces severe internal memory fragmentation; transitioning to fine-grained, dynamic paged allocation recovers substantial memory capacity [64], while alternative operating-system approaches preserve virtual contiguity via low-level memory primitives [98]. Furthermore, dynamic KV-cache prefix reuse needs to be managed opportunistically at runtime [75] across evolving serving systems and wafer-scale hardware substrates [80, 51].

These workload domains place two distinct requirements on the underlying memory subsystem. First, large, sparsely traversed working sets necessitate a compact translation scheme that has a large translation reach without proportional increases in translation metadata. Second, concurrent, fine-grained, and skewed allocation patterns require memory allocators that can dynamically balance loads. The translation mechanism that addresses the first requirement is developed in Chapter 4, while the allocation substrate that addresses the second requirement, which forms the management layer, is described in Chapter 5.

2.3 Global Memory Architecture

Modern high-performance computing and datacenter workloads increasingly exceed the memory capacity and bandwidth limits of single nodes, accelerating the transition toward scale-out, heterogeneous memory systems. High-throughput interconnect fabrics—utilizing RDMA over converged networks [119] or emerging Compute Express Link (CXL) standards—enable physical memory pooling across nodes, forming the architectural foundation for contemporary supercomputing systems [91, 9, 103] and specialized network topologies [66].

Contemporary architectures address the distributed memory wall through three dominant architectural paradigms, each imposing distinct constraints on address translation:

Hardware-coherent rack-scale integration. This approach extends hardware cache coherence across multi-node topologies. High-bandwidth interconnects such as NVLink and NVSwitch fabrics [89, 85], paired with tightly integrated CPU-GPU architectures such as Grace Hopper [87], synthesize multiple physical nodes into a unified, coherent address do-

main [88, 86]. However, empirical characterizations of deployed Grace Hopper systems indicate that inter-node data movement remains the dominant performance bottleneck despite transparent hardware coherence [39].

Compiler-driven deterministic execution. This paradigm removes dynamic memory arbitration from the execution path by enforcing static scheduling. Tensor Processing Units (TPUs) orchestrate data movement deterministically at compile time, managing on-chip SRAM explicitly without dynamic demand translation [56, 55, 44]. Deterministic dataflow engines [1], reconfigurable spatial architectures with dense on-package memory [97], and domain-specific accelerators [53] adopt similar static models. While effective for workloads with predictable communication patterns, this approach cannot accommodate the dynamic, data-dependent access patterns detailed in Section 2.2.2.

Near-memory event-driven execution. This architecture dispatches computation directly to target memory domains, utilizing lightweight, fine-grained events as the fundamental unit of execution. The UpDown architecture instantiates high-density, lightweight execution lanes situated near physical memory, with each lane issuing word-granular memory operations against a globally shared address space [100, 25] and integrating address translation with explicit locality management [116]. Figure 2.4 shows the architecture of a node for the UpDown system.

The system model adopted in this work builds upon this near-memory execution paradigm. The target architecture comprises thousands of execution elements providing word-level access to a shared global memory across a low-latency, high-bandwidth interconnect. All memory accesses, including local DRAM operations, are resolved through an integrated translation path. The resulting global address space is structured as a **2-dimensional space**: addresses are represented as coordinate pairs of the form `(node, offset)` rather than flat physical offsets. The goal is to preserve the physical data layout as a first-class property of the address representation and enable programmer-controlled, translation-governed data

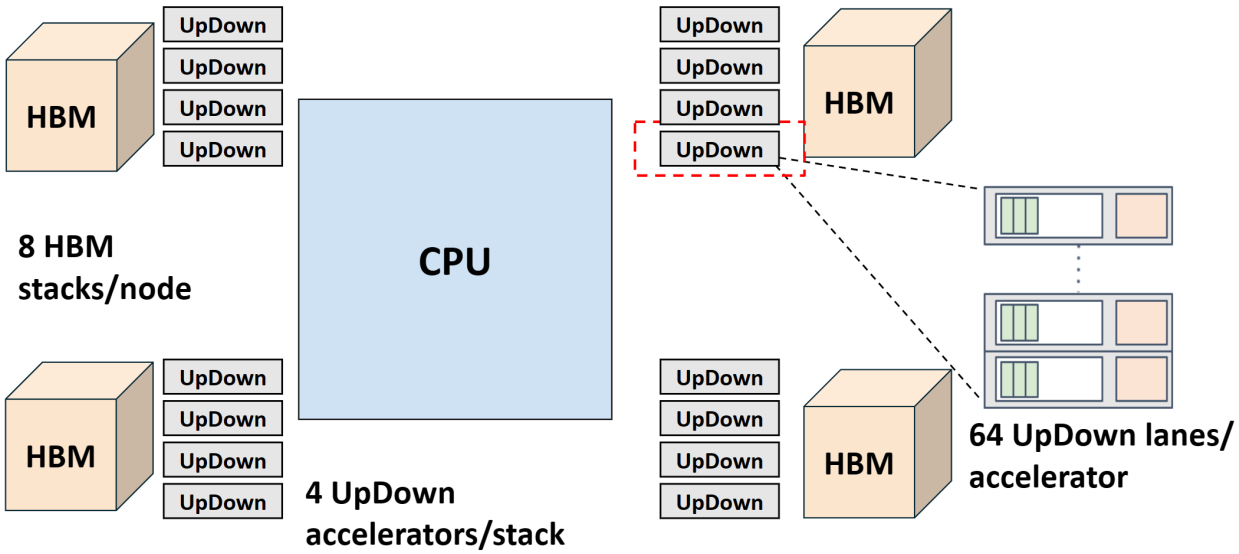


Figure 2.4: UpDown architecture (single node).

placement. Section 3.2.1 in Chapter 3 gives a precise definition of the 2D address space and the abstract system model we assume for the rest of the dissertation.

Address translation is a foundational architectural requirement for distributed global address spaces. Specifically, the translation subsystem needs to provide sub-linear metadata scaling with respect to mapped capacity, enforce memory protection across concurrent tenants, and enable software-guided data placement without privileged kernel intervention on the memory access path. The subsequent chapters detail the hardware translation mechanisms and runtime allocation substrates designed to meet these requirements.

CHAPTER 3

RESEARCH PROBLEM AND APPROACH

In the previous chapter, we defined the research context and motivated the global memory management problem for future scale-out distributed memory systems. This chapter provides an overview of this dissertation, including the research questions and the approaches we proposed to address the problem.

3.1 The Core Question

Managing a robust global memory for massive-scale distributed architectures is challenging mainly because of system control and access to shared memory resources. We divide the problem into two parts:

1. **Scalable Address Translation:** A virtual memory abstraction that guarantees hardware-level protection and efficient address translation for massive concurrent memory accesses over petabytes of memory.

2. **Global Memory Management:** A scalable software control plane that handles system-wide process isolation and coarse- and fine-grained memory allocation/deallocation with flexible data layout control.

These two components are closely related, and co-designing them allows the data layout to be expressed through allocation parameters and carried through to the translation layer. Memory management sets up the data layout and partitions physical resources for isolation and locality; address translation dynamically validates and routes every memory access to that layout at runtime, enforcing the layout without stalling execution. This approach enables programmer-controlled locality at a global level, addressing both performance and flexibility.

The solution should satisfy the following performance properties to be effective at the target machine scale:

- **Efficiency:** The design should minimize the size of the address mapping representation and reduce the "hot state" required in hardware translation units so they can fit within tight area and power constraints.
- **Scale:** It should cleanly separate protection (enforced in hardware) from management (handled by software). This decoupling ensures that memory operations remain scalable and load-balanced without relying on a centralized manager bottleneck.
- **Locality Control:** It should expose the distributed memory topology to software, empowering parallel applications to express and control their own data layout to exploit locality and bandwidth.
- **Parallelism:** It should handle highly parallel requests (billions or trillions per second) generated from millions of parallel processors concurrently.

3.2 Address Translation for Global Distributed Memory

3.2.1 Problem Definition

In conventional systems, the memory management unit (MMU) validates and translates every access to global memory, providing security, isolation, and programmability for system and application programmers. When a processing unit accesses a memory location, the translation logic validates the process's permission for the address and, based on the mapping, translates it into a physical address in the distributed global memory, ensuring secure access to shared memory under multiprocessing.

A conventional shared memory system assumes each application owns a contiguous virtual address space and translates addresses into a physical address in a single, flat, one-

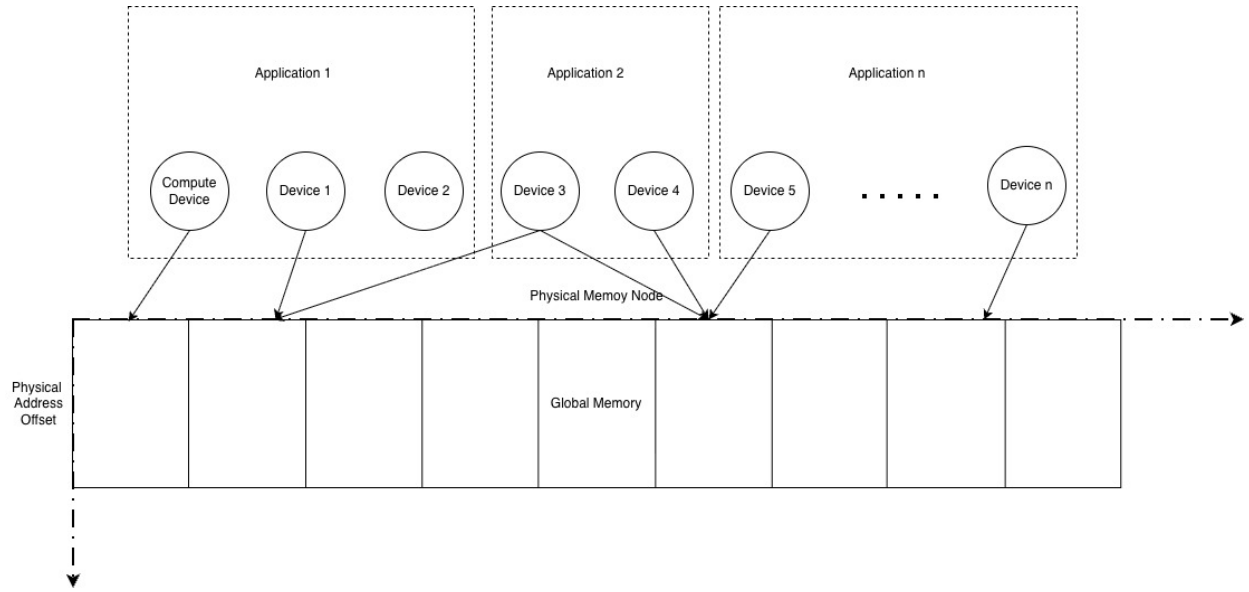


Figure 3.1: Memory Sharing over a 2-Dimensional Space.

dimensional (1D) physical address space, hiding the underlying memory hierarchy from software. In a distributed global memory system, however, physical memory experiences NUMA effects due to network topology, and the latency/bandwidth gap between local and remote memory can differ by several orders of magnitude. Therefore, we define the system's physical address to be 2-dimensional: (1) the system's memory node ID which holds the data, and (2) the offset within that node. We call this address space a 2-dimensional (2D) address space (Figure 3.1). Making the node dimension explicit and visible to software lets an application reason about, and ultimately control, the data layout (where its data is placed across the machine), which is critical for locality and performance in scale-out systems.

Address Translation Problem

Given a contiguous virtual address space `VA` for each application process and a global physical address space `PA` spanning the distributed memory nodes, define a mapping function F that maps every virtual address in `VA` to a unique physical address in `PA`. F, together with processor or system-specific state (e.g., base addresses, page/block size), translates every valid address

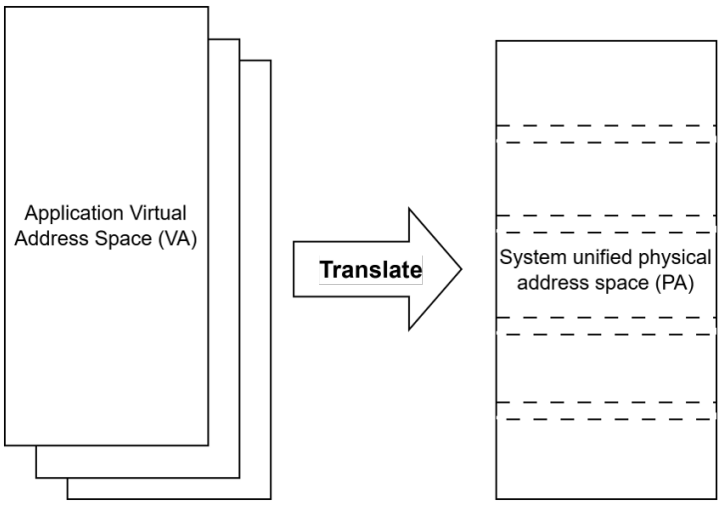


Figure 3.2: Address Translation Problem: Given an address `va` in `VA`, translate it into a unique address `pa` in `PA`. The physical address space consists of the memory of many distributed memory nodes, forming holes in the address space.

in `VA` to a valid address in `PA`. The mapping from `VA` to `PA` is one-to-one; the reverse mapping from a `PA` back to `VA` may be many-to-one across processes, so that a single physical location can be shared by, and mapped into, multiple processes, but has to be unique within a process's address mapping.

System model

To constrain the problem, we model the target machine as thousands of independent compute and memory nodes connected by a low-latency, high-bandwidth global network, as shown in Figure 3.3. Each compute node contains numerous lightweight parallel execution units, providing high-performance computing capability at scale. All processing units have high-rate word- and block-level access to a physically distributed global memory connected by low-latency, low-diameter networks. Together, the memory nodes form a petabyte-scale global memory, accessible by all processing units through a unified global physical address space.

This physical architecture resembles modern scalable systems, including the integrated

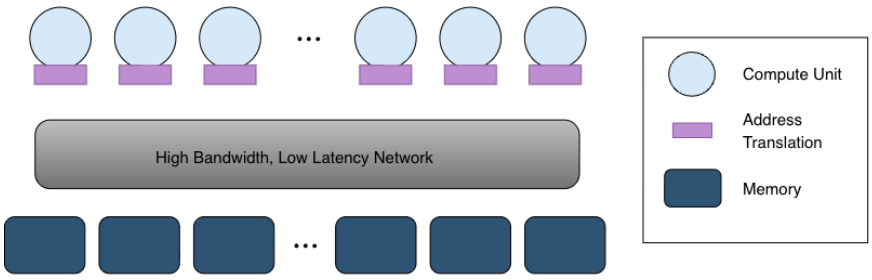


Figure 3.3: Basic Machine Model: Compute elements with word-level direct access to a global memory of petabytes large. All memory accesses, both to local and remote DRAM, are controlled by translation, a low-latency network (<1 us), with high bandwidth (PB/s) connects the machine.

Nvidia GPU supercomputer, Google TPU superpods, supercomputers, and research designs under IARPA's AGILE program [87, 56, 39, 45, 25]. With the 2D model we defined above, the node dimension of an address is instead a first-class, software-visible part of translation, so an application can control placement directly rather than relying on the system to guess.

We further relax the assumption that processing units are traditional load-store cores with a data cache; for example, future cache-less, customized accelerator cores [99, 114, 102]. In such systems, the number of in-flight asynchronous memory requests is no longer bounded by hardware structures such as MSHRs (Miss Status Handling Registers), placing substantial demand on the address translation layer. The solution needs to address the scenario where each processing core can issue a large, unbounded number of non-blocking asynchronous memory access requests concurrently.

Challenges and Requirements

Realizing global address translation at scale is difficult for two reasons:

1. because local accesses are 10x-100x lower latency and higher bandwidth than remote accesses, the scheme should let applications control data placement to exploit locality for performance

2. as the machine has on the order of millions of parallel processing units, the scheme needs to sustain a high rate of address translations spanning a large address space concurrently using translation hardware with high area- and power-efficiency

Therefore, a successful design is one that (a) preserves the process isolation that virtual memory should guarantee, (b) exposes the node dimension of the address so applications can place data for locality, and (c) meets a target translation rate within a limited hardware budget, evaluated relative to conventional page- and segment-based schemes.

In short, the first half of the dissertation aims to solve two research questions:

1. Given the locality and rate-under-budget requirements above, what translation scheme structure achieves both, and how should it be implemented?

2. What hardware is required to sustain low-overhead, high-rate address translation that meets these requirements in a scale-out machine?

3.2.2 Approach

Our approach first abstracts address translation over a 2-dimensional address space into a **translation descriptor**, defines the design space for the translation descriptor, and positions the conventional page-based and segment-based address translation schemes based on their bottlenecks and limitations in a scalable context to identify the gap. We then propose a compact swizzle translation entry to fill this design-space gap: it combines the scalability of a segment-based scheme with the flexibility of distributing data across non-contiguous address spaces. This approach gives applications direct control over data distribution across the 2D address space. To meet the requirements defined above, we also propose hardware to accelerate this translation layer, with a translation cache analogous to a TLB in conventional page-based schemes, so that the common-case translation is fast without violating the hardware area/power budget.

To demonstrate that this design meets the requirements set out in Section 3.2.1, we evaluate it on a trace-based address translation logic simulator against the page-based and segment-based baselines along three axes tied directly to those requirements: (1) translation rate, measured as sustained translations per second per translation unit under the large-scale parallel, potentially unbounded-asynchrony access patterns described above; (2) hardware cost, measured as the area and power of the translation logic and translation cache relative to the baselines; and (3) locality, measured as achieved latency and bandwidth when an application places data using the swizzle scheme's software-visible layout control, compared to the same workload under a scheme that does not expose the node dimension. Matching or improving on the baselines along all three axes, using the same simulator and workloads, is what constitutes evidence that the requirements have been met.

3.3 Scalable Memory Management

In the second half of the dissertation, we target the global memory challenge from a different angle: maintaining a coherent, consistent, system-wide view of the global address space, and isolating each application's accesses to that shared resource, is costly at scale. Traditional shared-memory systems rely on centralized memory management to ensure secure access to global memory: a single, system-wide allocator services all memory allocation requests. In traditional translation schemes (page- or segment-based), a valid entry in the translation table is itself the evidence that access is permitted, so managing translations is necessarily a privileged operation requiring expensive OS kernel calls. This is feasible in moderate-sized commercial systems, given their limited memory capacity and parallelism. For a system with petabytes of global memory and millions of parallel processing units, however, a single centralized allocator becomes a performance bottleneck.

This bottleneck is compounded by a second issue: applications exhibit widely different memory access patterns and need different mapping structures to achieve good parallel per-

formance. Distributing data uniformly across the system's memory nodes, without regard to an application's own knowledge of its access pattern, substantially impairs performance given the order-of-magnitude difference in latency and bandwidth between local and remote accesses. A single system-level allocator therefore not only creates a bottleneck, but it also prevents applications from exploiting locality using their own, application-specific knowledge.

3.3.1 Problem Statement

The problem is to design a memory management scheme for a massive, distributed global memory that (1) removes the performance bottleneck inherent in a single, centralized allocator servicing millions of concurrent processing units, (2) safely isolates each process's allocations from every other process sharing the global memory, and (3) preserves each application's ability to control its own data layout for locality, consistent with the programmability requirement.

Answering this problem requires answering two sub-questions:

1. How to eliminate the bottleneck of having centralized memory management in a scale-out machine while guaranteeing that applications safely share global resources?

2. Can the scheme enable various applications to achieve data locality and parallel performance with different memory allocation policy?

3.3.2 Approach

We propose a two-level memory management scheme to address this problem. The first level is system-level privileged: a global manager performs large static chunky global allocation for each application. This partition gives each process a safe, isolated share of global memory, satisfying sub-question 1's isolation requirement. Because this step happens off the allocation hot path, it does not reintroduce the bottleneck of a single, centrally serviced allocator: every

fine-grained dynamic allocation and deallocation issued while the application runs is handled locally by the application at the user level. There is no global rebalancing step on the critical path, addressing the remaining sub-questions.

The second level targets the second sub-question: an application-local allocator lets the application choose a customized allocation algorithm and policy, not one the system fixes for it. A general-purpose backend (e.g., a buddy allocator) suits workloads with variable-sized, dynamically freed allocations; a backend specialized for many fixed-size records (e.g., a slab allocator) suits workloads dominated by one record size and benefits from the lower per-allocation overhead that specialization allows. The second-level manager with different backends sits on top of the isolated address mapping (segments) set up by the first level, so two applications sharing global memory can each choose the allocation policy suited to their access pattern without affecting the other's isolation guarantee. We implement and evaluate this scheme on a global-memory machine with million-fold parallelism and a unified global memory, using both regular and bursty concurrent allocation/deallocation workloads to stress the allocation and load-balancing paths. We will report both the steady-state throughput and the performance under skewed, imbalanced allocation request distributions.

To evaluate the user-level memory management under this scheme, the dissertation proposes a fine-grained, hierarchical memory allocator, consisting of both a buddy and a fixed-size slab backend. We evaluate the performance of that allocator on bursty and imbalanced workloads representative of real-world usage to demonstrate the importance of allocator exploiting application knowledge for load-balancing and parallel performance.

3.4 Summary

In summary, global address translation and memory management are two halves of the scalable global memory, combining to form a complete solution for distributed memory architecture. The two-level memory management scheme provides a highly concurrent and

flexible software control plane: Level-One divides the massive global memory into safe, application-managed partitions, avoiding bottlenecks from a central privileged allocator, and Level-Two enables programmer-controlled, application-specific data layout control and fine-grained memory allocation. Orthogonal to it, the swizzle translation provides efficient access to global memory, enforcing the isolation guarantees while concurrently translating and routing memory accesses from millions of parallel threads. Together, they address the core thesis of this dissertation: by decoupling system-level resource isolation from user-level locality management and co-designing the hardware translation to natively support this division, we can achieve a scale-out shared memory system that maximizes both high-throughput performance and developer programmability.

CHAPTER 4

UDTRANS: EFFICIENT ADDRESS TRANSLATION FOR DISTRIBUTED GLOBAL MEMORY

This chapter presents UDTrans, a compact address translation scheme with application-controlled data layout. It introduces an arbitrary-sized translation entry augmented with a 64-bit mask for precise data placement. This chapter details the translation structures and hardware design, and evaluates the scheme using a trace-driven translation simulator.

4.1 UDTrans Design

We propose the compact translation scheme, UDTrans, which expresses an application's address mappings using minimal translation state, thereby incurring negligible performance and resource overhead. This section is organized as follows: the formal definition of the 2-dimensional physical address space (Section 4.1.1), the design space of translation descriptors and the reversibility requirement (Section 4.1.2), the swizzle descriptor space (Section 4.1.3), the translation entry and table architecture (Section 4.1.4), and the corresponding hardware design (Section 4.1.5).

4.1.1 The 2-Dimensional Global Physical Address Space

The fundamental challenge in scalable systems is managing the physically distributed memory. To formalize the problem, we abstract memory into a 2-dimensional global address space that preserves the physical topology of distributed memories while providing a unified, globally shared address space across the system.

Every physical memory location within the target machine architecture is uniquely addressable via a 64-bit physical address. We define the upper 19 bits as the memory node identifier and the lower 45 bits as the byte offset within that specific node's memory, as illus-

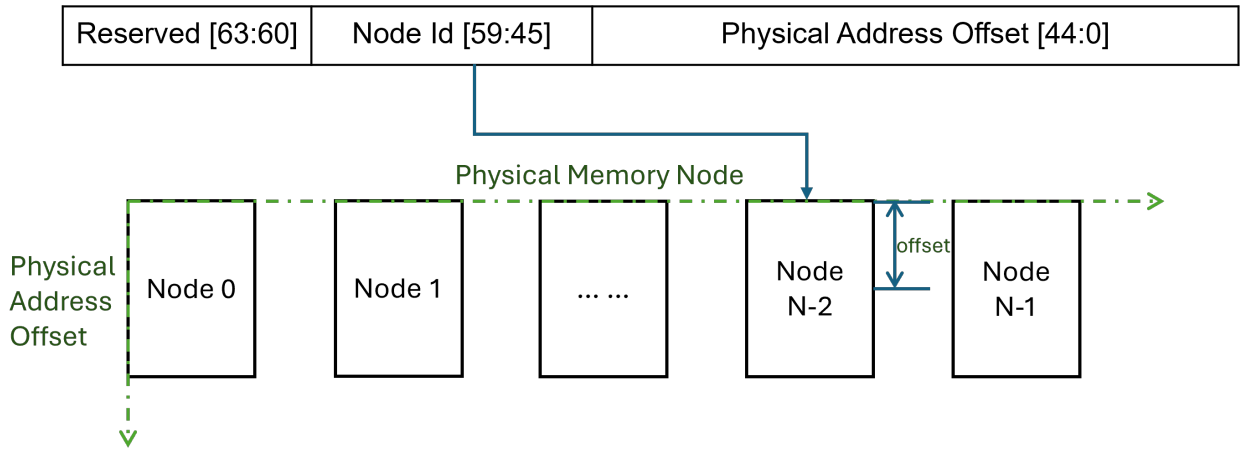


Figure 4.1: Physical address format (top) and the 2-dimensional physical address space (bottom).

trated in Figure 4.1. By construction, the physical address space is 2-dimensional: memory nodes represent one dimension, while the intra-node offset represents the other. Unlike a traditional flat physical address space, a 2-dimensional address space makes where a virtual region is placed a property that the translation scheme can manage explicitly.

4.1.2 Translation Descriptors: The Design Space and Reversibility

A translation descriptor consists of a mapping function `f` and the state `s` by which it is parameterized.

Given an address α, the descriptor (including mapping function f and states s) converts it to

$$\beta = f(\alpha, s) \tag{4.1}$$

where α is an address in the application's virtual address space, and β is the physical address it is mapped onto.

The design space of translation descriptors ranges from highly restricted and resource-efficient (e.g., segmentation) to highly flexible and resource-intensive (e.g., paging). A fixed-

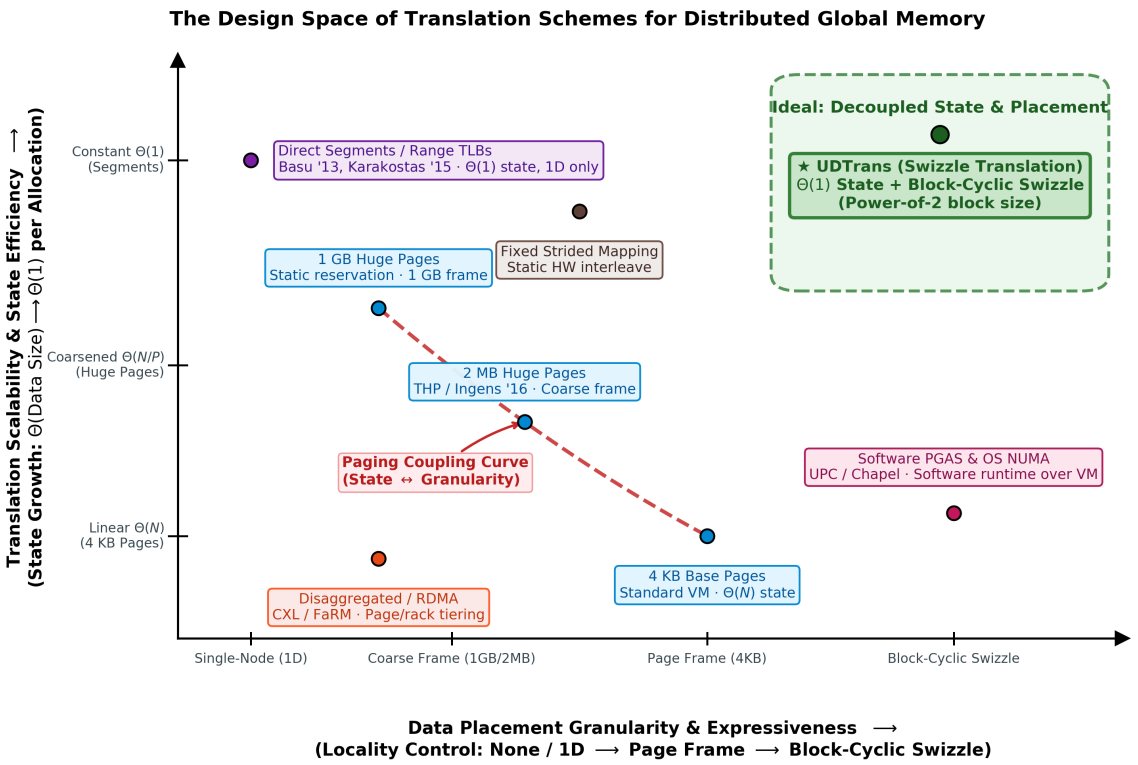


Figure 4.2: The Design Space of Translation Schemes for Distributed Global Memory

stride interleave requires negligible storage and overhead but cannot express non-contiguous layouts. Conversely, a descriptor that can split a region into arbitrarily placed, fixed-sized sub-blocks offers maximum flexibility but requires extensive state to track the mappings, i.e., a conventional page table with 4 KB pages. Intermediate solutions include direct mapping (a single segment per application: minimal computation, minimal expressiveness), reversible hash-based translation, and structured-set families that map consecutive inputs onto a discrete space. Figure 4.2 illustrates the design space of address translation schemes and how UDTrans fills in the gap in the space that scalable distributed memory systems demand.

Two criteria guide descriptor selection within this space. The first axis is **expressiveness**: the descriptor should be flexible enough to express a program's data placement based on its access patterns, yet compact enough that storing the descriptor itself does not become a bottleneck. The second is **efficiency**, measured by the descriptor states (metadata for

bookkeeping) required to cover an application's valid address mappings (i.e., working set). Page-based translation is inefficient because the translation state (the Page Table) scales proportionally to data size. In contrast, segmentation's single-segment entry can describe an arbitrarily large region, which is highly efficient and compact in translation state, but requires contiguity in physical memory. Because translation state size affects the scheme's performance overhead, a scalable descriptor should cover an application's address mappings with a minimal set of states.

A third aspect is **reversibility**: A translation unit should not only map virtual addresses to physical addresses but also reverse-translate physical addresses when memory requests return from DRAM. The literature often omits reverse translation because traditional schemes map one contiguous space onto another, and caches with MSHRs track this mapping during a memory access. However, UDTrans maps consecutive virtual addresses onto disjoint physical regions and targets broader cache-less architectures where this problem is opaque, so it is restricted to bijective mapping functions. This ensures that reverse translation requires no additional hardware resources beyond those used for forward translation.

Finally, multiprocessing is supported in a conventional way: a process identifier is included in the descriptor state, enabling separate processes to map distinct virtual addresses to the same physical location securely.

4.1.3 The Swizzle Descriptor

UDTrans proposes a new compact descriptor, the swizzle descriptor, that compactly expresses the block-cyclic data layouts prevalent in distributed-memory workloads. The descriptor includes the base and bound addresses to specify the address range it covers and a 64-bit swizzle mask consisting of alternating runs of zeros and ones to express the distribution of consecutive addresses across memory nodes. The four constituent digits of this mask each serve a function in expressing block-cyclic distribution (Figure 4.3). Let P, F, B, and C

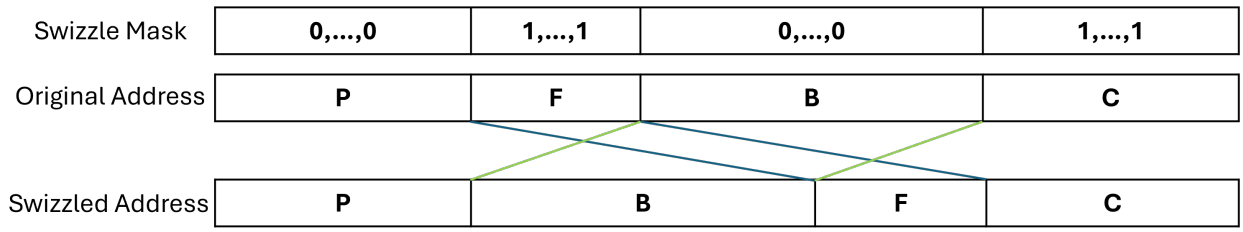


Figure 4.3: Swizzle mask (top) and the corresponding address-swizzle operation (bottom).

denote the bit-widths of these alternating runs from the most significant bit downward. C represents $\log_2$(block size); B represents $\log_2$(number of memory nodes the segment spans); F acts as padding such that F + C equals $\log_2$(memory per node); and P pads the mask to a full 64 bits. When constructing a mask for a given allocation, the user specifies the block size and node count (C and B), while the machine's physical memory configuration dictates the remaining parameters.

The swizzle operation transposes the address bits under F with the bits under B. Once this swizzled offset is added to the physical base, the bits at positions P and B designate the physical memory identifier, while the bits at F and C dictate the byte offset within that specific memory node. B and F do not need to be equal for this transposition to be well-defined. The architectural effect is that the C-width lower bits of a block number serve as the intra-block offset, while the subsequent B-width bits cyclically select the participating node on which the block resides. Therefore, consecutive blocks in the virtual address space land on consecutive memory nodes, wrapping around once they reach the boundary node. The descriptors' physical base defines the node where the first block is placed; the mask expresses the shape of the distribution. This operation is a bit permutation and, hence, is bijective by construction and satisfies the reversibility requirement outlined in Section 4.1.2. The same logic applies to both forward and reverse translations, mapping contiguous linear virtual addresses to non-contiguous block-cyclic physical addresses (Figure 4.4).

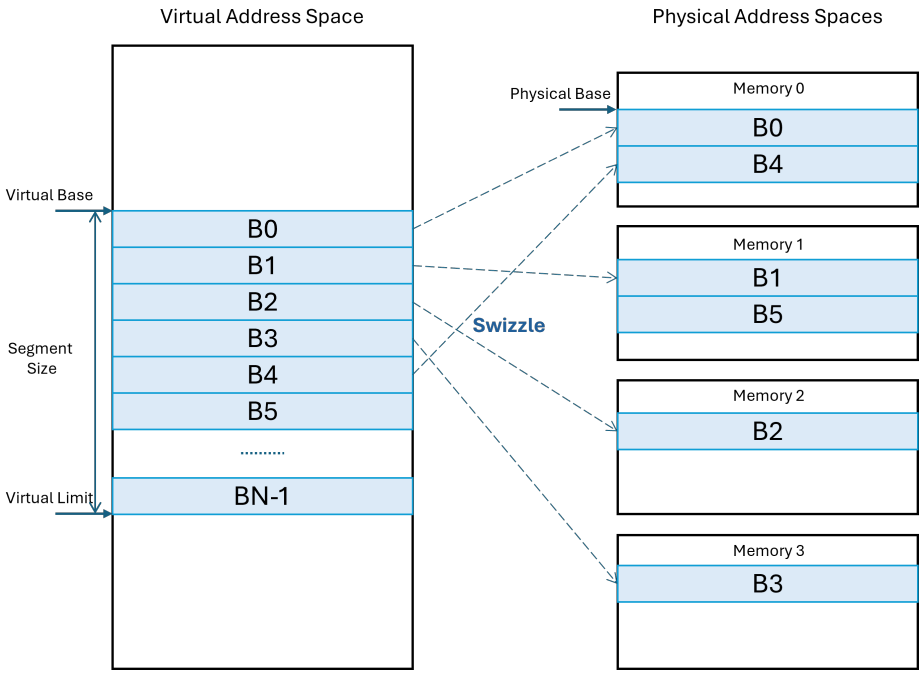


Figure 4.4: Each translation entry expresses a swizzle mapping between the virtual and physical address spaces. Contiguous address blocks in the virtual address space are mapped cyclically to blocks distributed across memory nodes.

4.1.4 Translation Entry and Translation Table

UDTrans' translation entry is structurally analogous to a traditional segment: Each entry contains a base virtual address, a limit defining the region's size, a physical base address, permission bits, and the swizzle mask (Figure 4.5). Thus, a single entry can map an arbitrarily sized region.

An application's complete set of translation entries is maintained in a **UDTrans table**, the analog to a page table or segment table. Because each entry covers an arbitrarily large region, this table needs orders of magnitude fewer states than a traditional page table to express address mapping for the same-sized memory region. This compactness yields two key architectural benefits: 1) the table can be replicated across distributed memory nodes with low communication and synchronization cost, and 2) it can be stored in shared SRAM between compute elements rather than in slower DRAM. While replicating translation entries

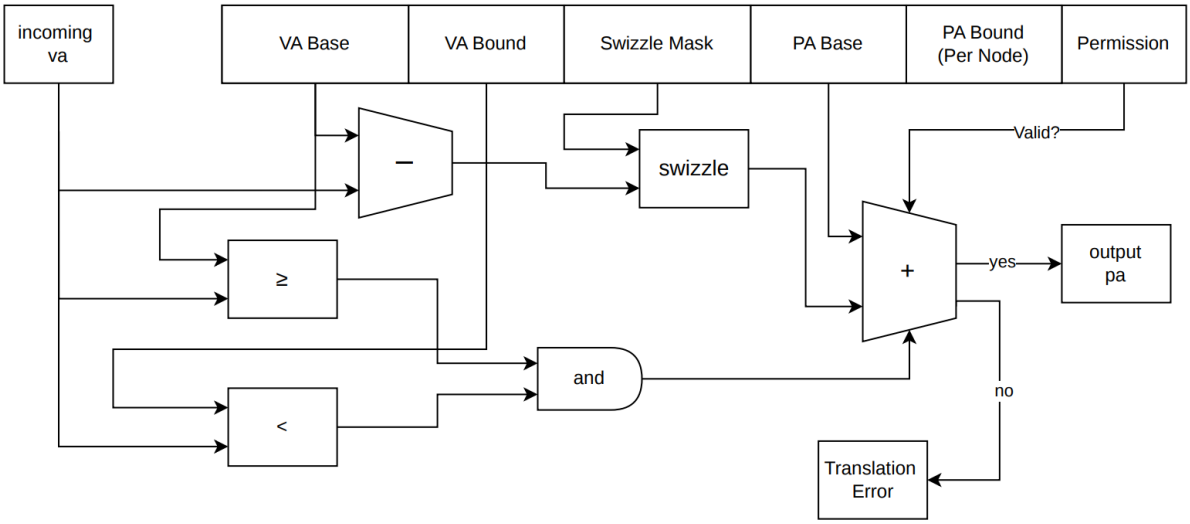


Figure 4.5: A UDTrans translation entry featuring the swizzle mask, alongside the hardware logic that executes the translation.

across the system necessitates coherence protocols, the associated overhead is negligible because each entry is bounded to 48 bytes.

The table is structured as a binary search tree to reduce the overhead of linear scan on every memory access: addresses below an entry's base traverse the left subtree, and those above its limit traverse the right subtree (Figure 4.6). This yields lookup complexity logarithmic to the number of entries, and any mismatch triggers an access violation. This approach is efficient because UDTrans targets data-intensive applications that allocate massive datasets with few discrete calls. Furthermore, because a UDTrans segment need not be contiguous in physical memory, it is not constrained by a single memory node's capacity. Therefore, applications are expected to require only a minimal number of entries to express all their translation mappings. In Chapter 5, we explore whether this assumption constrains address mapping flexibility and detail how our software approach resolves potential limitations while minimizing translation overhead.

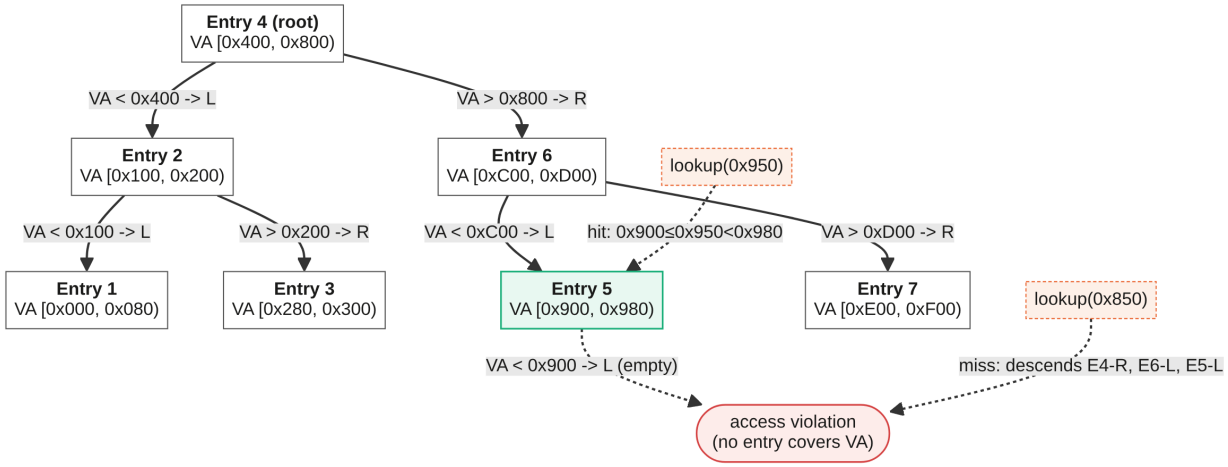


Figure 4.6: UpDown Translation Table: binary lookup by the virtual addresses

Compatibility in Accelerator-Based Systems

In heterogeneous accelerator-based systems, modifying the host CPU's page-based translation hardware and software stack incurs substantial complexity. However, allowing the accelerator to leverage UDTrans's compactness and scalability remains advantageous. To ensure compatibility, we propose a scheme in which two independent hardware contexts (one on the host CPU and one on the accelerator) map addresses to the same physical substrate, maintaining a coherent view of the global address mapping.

The host CPU continues to use conventional page-based address translation with standard TLBs and page walkers, requiring no hardware modifications. When the accelerator requires access to shared memory, the system installs page table entries that express the exact physical-to-virtual (PA-to-VA) mappings corresponding to the UDTrans entries upon allocation. The accelerator concurrently uses UDTrans's native structures, which can support far more concurrent threads than a CPU, ensuring that per-device translation structures remain substantially smaller and faster than a conventional TLB.

4.1.5 Translation Hardware

A software-only implementation of the swizzle descriptor would incur performance overhead (memory latency for address translation); therefore, we propose dedicated hardware logic to perform translation, analogous to the Memory Management Unit (MMU) on a standard CPU. The principal constraint is the physical hardware budget, and the area allocated for translation is limited, bounding both the metadata stored near the compute lanes and the complexity of the mapping logic itself. The logic needs to service concurrent translation requests from massively parallel arrays at high throughput.

UD-TLB

To remove table lookups from the critical path, frequently accessed entries are cached in a compact, fully associative array featuring single-cycle access latency, designated the UD-TLB (Figure 4.7). The baseline design uses 8 entries, each requiring only 64 bytes (512 bits), with a single array shared across 4 compute lanes. This configuration ensures that lanes issuing memory references every 4 cycles will not stall. A cache miss forwards the request to the table walker; upon retrieval, the new entry refills the array, evicting the least recently used entry when at capacity. We treat the entry count as an evaluated parameter rather than a fixed architectural assumption throughout Section 4.3, since the sufficiency of an 8-entry cache is intrinsically tied to the software stack's allocation behavior.

Translation memory

A copy of the UDTrans table is retained in SRAM shared across the processing cores. As we discussed in Section 4.1.4, a table with tens of entries is sufficiently large under our allocation assumptions (for instance, similar to or smaller than the size of the Page Walk Cache) to serve DRAM lookups during a UD-TLB miss. The memory is sized at 16 KB, accommodating 256 entries (each comprising a 48-byte payload plus tags, flags, and alignment padding). A full

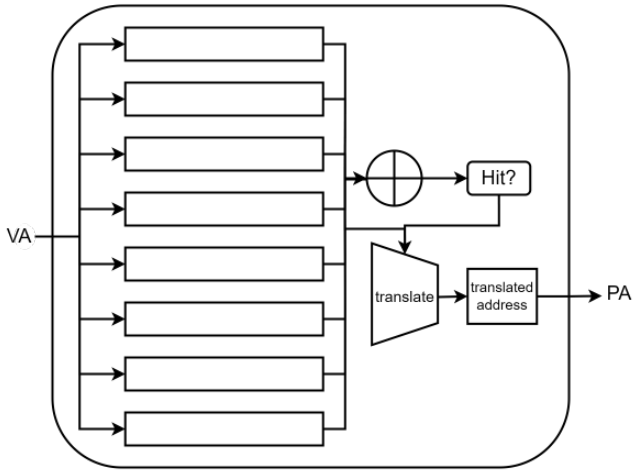


Figure 4.7: The UD-TLB: a compact, fully associative translation cache with single-cycle access latency, shared by four compute lanes.

table retrieval requires up to 9 SRAM lookups in the worst case (a full binary tree traversal of depth 8, including the root).

Power model

Both the UD-TLB and translation memory components are modeled using CACTI [81], which provides validated estimates for timing, dynamic energy, and leakage for SRAM and cache structures as a function of technology node, geometry, and associativity. We model both components as fully associative structures at the 22 nm node, mirroring a realistic RTL implementation.

4.2 Methodology

4.2.1 Simulator and Metrics

We evaluate the proposed and baseline translation mechanisms using a custom, trace-driven C++ simulator that accurately models the hardware translation paths and cache hierarchies. We replay each trace record against the respective scheme's logic, dynamically updating TLB

and walk-cache states according to hardware specifications and registering a hit when the descriptor or translation entry is present in the TLB. We report four core evaluation metrics across all workloads and configurations:

- **TLB Hit Rate**: The fraction of memory translation requests resolved by the first-level TLB without initiating a page-table or segment-table walk.
- **Memory Access Amplification**: The additional DRAM accesses incurred per application memory request to resolve address translation misses, expressed as a percentage.
- **Runtime Overhead**: The fraction of execution time attributable to address translation under a throughput-constrained memory model.
- **Power**: Total translation power dissipation, modeled using CACTI at the 22 nm technology node for on-chip SRAM/CAM arrays combined with the dynamic energy consumed by translation DRAM traffic.

4.2.2 Memory Trace Collection

We evaluate workloads across two distinct sets: a conventional CPU/GPU trace set and the trace set executed by the UpDown accelerator architecture. We analyze these traces separately because they capture different abstraction levels: CPU/GPU traces record source-level element references, whereas UpDown traces capture direct, word-level memory requests issued by accelerator lanes to DRAM.

CPU/GPU Traces

The first trace suite (Table 4.1) collects source-level memory accesses from diverse parallel benchmarks and applications: distributed MPI programs (including PGAS, GUPS, and variable-segment microbenchmarks; FFT, SpMV, Stencil, BFS, and PageRank kernels; and

NAS Parallel Benchmarks (NPB) CG and IS at Class E) executed across $N \in \{64, 128\}$ ranks with memory footprints ranging from 78 GB to 565 GB; shared-memory graph algorithms from the GAP Benchmark Suite (GAPBS) and Monte Carlo transport simulations from XSBench under native instrumentation; and a GPU-accelerated BFS kernel captured via NVBit on an NVIDIA A40 GPU. Because these source-level traces lack explicit hardware cycle counts, we assume a normalized baseline of 2 ns per reference for power and throughput calculations.

UpDown Traces

The second trace suite (Table 4.2) is generated using the cycle-accurate UpDown architecture simulator, which models a massively parallel, distributed near-memory compute architecture. The full UpDown system scales up to 16,384 nodes (each comprising 2,048 MIMD lanes), providing 2 PB of global physical address space and 30 TB/s of aggregate memory bandwidth per node, with each lane capable of issuing a word-level reference every 4 cycles. We evaluate representative workloads on system configurations from 2 to 128 nodes, holding each application's memory footprint fixed so node count is the only variable. Because these traces record exact kernel cycle counts, we calculate performance and power metrics from reported runtime.

4.2.3 Configurations

We evaluate four address translation schemes across identical memory access streams (Table 4.3): (1) a conventional 5-level radix page table with 4 KB pages (representative of x86-64 LA57), (2) a 4-level radix page table with 2 MB huge pages, (3) a flat per-node segmentation baseline, and (4) UDTrans.

The segmentation baseline models conventional base-and-bound descriptors applied to distributed memory: distributing a data structure across N nodes requires N discrete seg-

Table 4.1: Workload traces collected from CPU/GPU. Total trace memory accesses (N) range from 0.07 G to 22.44 G operations across workloads. All CPU traces use $P \in \{64, 128\}$ ranks except natively instrumented GAPBS and XSBench (P=1); GPU trace uses a single NVIDIA A40 device. Traces follow the unified allocation model (one contiguous rank share per node).

Workload	Data Structs	Footprint	Access Pattern
PGAS Microbenchmarks			
`blocked`	1 array	128 GB	Contiguous rank-local reads
`cyclic`	1 array	128 GB	Element-cyclic interleaved reads ($r \equiv \text{rank} \pmod P$)
`halo`	1 array	128 GB	Blocked reads with 10% nearest-neighbor boundary sharing
`random`	1 array	128 GB	Uniform-random array reads
`mixed`	2 arrays	128 GB	Alternating blocked and cyclic array reads
Variable Segment Microbenchmarks			
`GUPS`	1 table	128 GB	HPCC RandomAccess: uniform-random atomic RMW updates
`RadixSort`	2 arrays	128 GB	Radix-256 scatter pass across 2 packed segments
`VarSeg`	S arrays	128 GB	Blocked round-robin access over $S \in [8, 64]$ arrays
MPI Application Workflows			
`FFT`	4 structs	206 GB	Radix-2 2D FFT: strided butterfly and transpose stages
`SpMV`	5 structs	128 GB	Sparse matvec (CSR): indirect vector gathers at column indices
`Stencil`	2 structs	256 GB	2D 5-point nearest-neighbor stencil sweeps
`BFS`	3 structs	78 GB	Level-synchronous top-down BFS with post-exchange updates
`PageRank`	5 structs	82 GB	Pull-model PageRank: gather contributions from in-neighbors
NPB Class E			
`CG-E`	8 structs	$\sim$110 GB	Conjugate gradient: sparse matvec with randomized sparsity
`IS-E dyn`	11 structs	$\sim$565 GB	Integer bucket sort with dynamic bucket scheduling
`IS-E cyc`	11 structs	$\sim$565 GB	Integer bucket sort with static cyclic bucket scheduling
Graph Analytics, Monte Carlo & GPU Benchmarks			
`GAPBS-kron`	4 structs	$\sim$155 GB	BFS on Kronecker graph (P=1)
`GAPBS-urand`	4 structs	$\sim$155 GB	BFS on uniform-random graph (P=1)
`XSBench`	5 structs	68 GB	Monte Carlo neutron cross-section hash lookups
`cuGraph-BFS`	51 structs	51 GB	GPU-accelerated BFS neighbor traversal

Table 4.2: Application traces collected from the UpDown architecture simulator (BASim). Each application is evaluated with its memory footprint held fixed across node counts $N \in \{2, 4, 8, 16, 32, 64, 128\}$ (2,048 lanes/node), rendering trace volume invariant in N. Each data structure corresponds to one UDTrans global memory entry.

Workload	**Data Structs**	**Footprint**	**Access Pattern**
`SpMV`	7 structs	64.0 GB	Sparse matvec ($y = Ax$, MSR format): device-issued indirect vector gathers of x at column indices.
`FFT`	3 structs	34.4 GB	Radix-2 2D FFT: device-issued strided butterfly and transpose stages.
`PageRank`	5 structs	39.2 GB	Push-split PageRank (KV map-reduce): scatter per-vertex contributions to out-neighbors.
`BFS`	6 structs	37.3 GB	Dynamic-array frontier expansion: device-issued neighbor-list walks across active vertices.

Table 4.3: Simulated translation schemes. All four schemes see the identical access stream from each trace; the segment arm's per-node split uses the trace's own rank count N.

Scheme	**Translation Mechanism**	**TLB Config**	**Cache / Hardware Structures**
Page (4 KB)	5-level radix page table (LA57)	256-entry (64 sets × 4 ways)	Page-walk caches on PGD / P4D / PUD / PMD
Page (2 MB)	4-level radix page table (LA57)	256-entry	Page-walk caches on PGD / P4D / PUD
Segments	Flat base/limit with per-node descriptors (N slices per alloc, slice k on node k)	8-entry fully-assoc segment TLB	128-entry (32 × 4) descriptor cache
UDTrans	1 descriptor per allocation + 8 B block-cyclic swizzle mask	8-entry UD-TLB	16 KB, 256-entry UTM SRAM (48 B payload)

ment descriptors per allocation. This baseline is provisioned with an 8-entry segment TLB and a 128-entry descriptor cache backed by DRAM fetches on misses.

Furthermore, we map legacy MPI traces into a unified Partitioned Global Address Space (PGAS). In conventional MPI executions, each rank invokes private memory allocators within isolated address spaces, instantiating disjoint memory slices for distributed data structures (e.g., five distributed structures across four ranks yield twenty isolated allocations). This disjoint allocation model is an artifact of legacy distributed-memory runtime and misaligns with UDTrans's global allocation model, where a single logical data structure spans multiple nodes via hardware swizzle masks. Evaluating UDTrans on ununified rank-private allocations would artificially fragment translation entries. To ensure a fair and architecturally sound comparison, we re-instrumented multi-rank MPI traces into a unified global address space: each distributed structure is represented by a single global declaration with power-of-two padded rank strides (forming the swizzle mask), and all rank-local addresses are remapped to a contiguous global window. We discuss the implications of power-of-two stride alignments in Section 4.3.3.

4.3 Evaluation

Our evaluation centers on two central research questions:

1. What are the translation overheads, in terms of TLB hit rate, memory access amplification, runtime overhead, and power consumption—across conventional CPU/GPU and accelerator workloads (Sections 4.3.1 and 4.3.2)?

2. What is the architectural cost required to distribute data across multiple memory nodes, which motivates UDTrans's swizzle mask design (Section 4.3.3)?

4.3.1 Performance on CPU/GPU Traces

TLB Hit Rate

Figure 4.8 presents the mean TLB hit rates per framework. At memory footprints of 64 GB and above, 4 KB paging experiences severe TLB reach thrashing: 28 of the 46 workloads exhibit hit rates below 1%, including all PGAS, GUPS, VarSeg, RadixSort, SpMV, BFS, and PageRank. The hit rate drops significantly because a 256-entry TLB covers only 1 MB of address space, five orders of magnitude smaller than the gigabyte-scale active working sets. The 4 KB configuration achieves moderate performance only on workloads with dense spatial locality: natively instrumented GAPBS (78.4% mean), XSBench (89.3%), and the GPU capture (75.5%). Expanding page sizes to 2 MB extends TLB reach to 512 MB, recovering hit rates on locality-friendly workloads (XSBench: 97.4%, cuGraph: 99.1%, GAPBS: 84.6%, NPB: 50.7%) while remaining below 0.5% for irregular-access applications. Per-node segmentation's TLB residency is far more workload-dependent: requiring $N \in \{64, 128\}$ descriptors per allocation overwhelms the 8-entry segment TLB on six of the nine frameworks (mean hit rates at or below 15.8%), but GAPBS (60.7%), XSBench (42.0%, where its 48-byte `num_nucs` data structure is installed as a single whole descriptor without splitting), and cuGraph (99.0%) retain far higher residency thanks to smaller per-node descriptor working sets. Conversely, UDTrans maintains hit rates exceeding 99.99% across 33 of the 46 workloads. Its lowest hit rates occur where round-robin traffic spans more segments than the 8-entry UD-TLB holds: NPB's IS-E bucket sort, i.e., across 131 declared bucket structures, falls to 85.6%, and VarSeg's $S \geq 16$ arrays at $N = 64$ reach 98.44%. In both cases, misses are all served by the 256-entry translation memory.

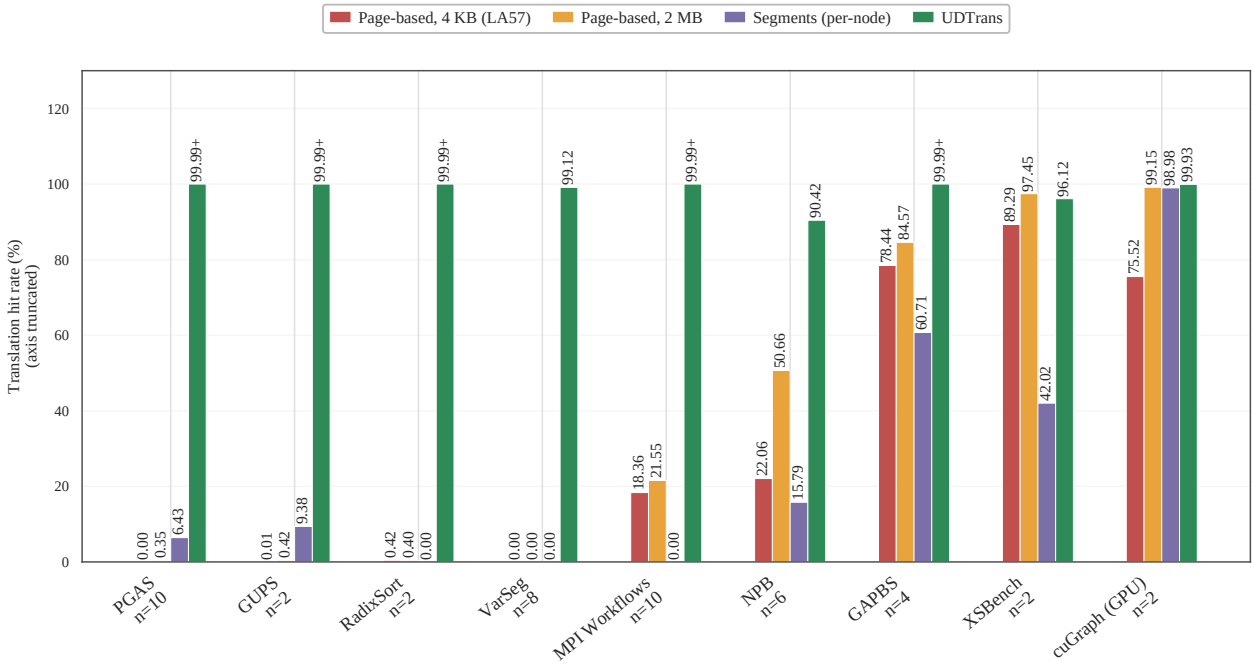


Figure 4.8: TLB hit rate across the CPU/GPU traces (per-framework arithmetic mean). Per-node segmentation is undefined (n/a) on single-node traces.

Memory Access Amplification

Translation-miss penalties vary substantially across architectures. Conventional 4 KB paging incurs a geometric-mean memory-access amplification of 164.8% across the workload's total DRAM traffic, with a peak amplification of 298%. This occurs because multi-gigabyte working sets evict both the lower- and upper-level page table entries from page walk caches. Figure 4.9 expresses the other three schemes' DRAM traffic as a multiple of this 4 KB baseline (ratio of total traffic). 2 MB huge pages reduce DRAM traffic by only 1.1-1.5× per framework, since the overhead from irregular access patterns persists at the larger granule size. Per-node segmentation's ratio is workload-dependent (1.0-3.6×): it reaches 3.6× on PGAS, where its descriptor working set ($N \cdot D$) largely fits within the 128-entry on-chip descriptor cache, but drops to 1.5× on VarSeg ($N \cdot D \in [512, 8192]$), where the descriptor cache suffers severe thrashing, and translation lookups reach DRAM (100% amplification). In contrast, UDTrans achieves zero DRAM access amplification across all 46 traces, as all

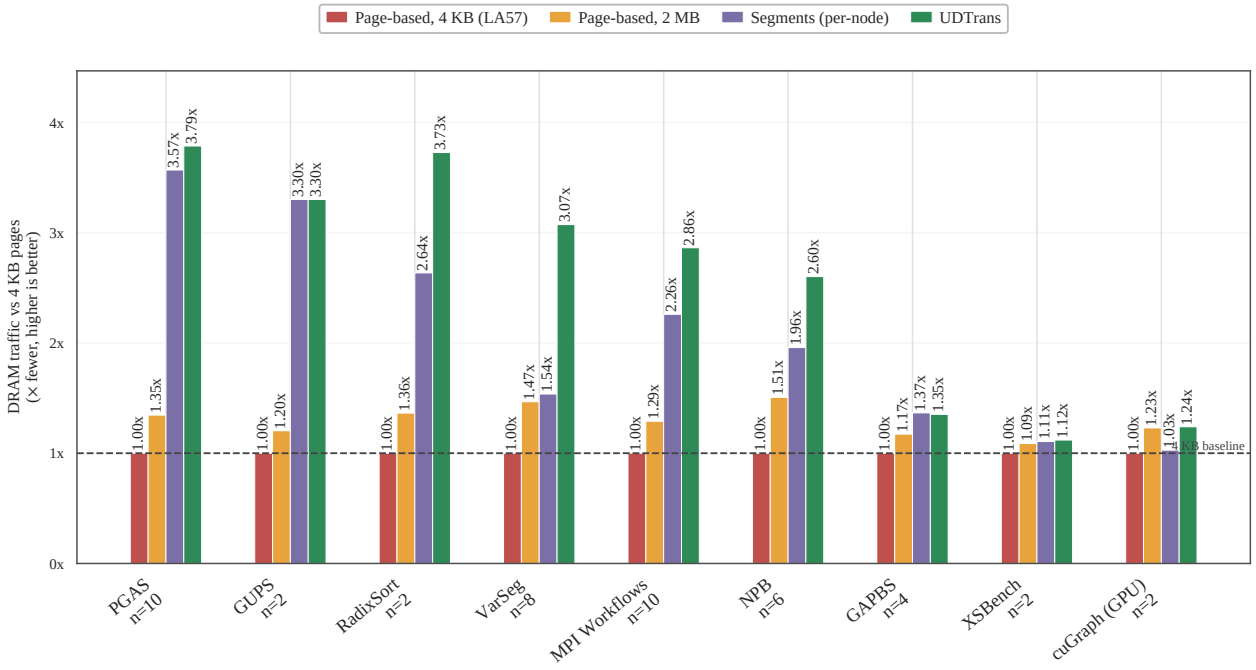


Figure 4.9: DRAM traffic relative to 4 KB pages on the CPU/GPU traces (each bar is a per-framework geometric mean of the total-traffic reduction ratio).

secondary lookups are resolved within on-chip SRAM; its ratio (1.1-3.8× per framework) tracks how amplification-prone 4 KB paging was on that framework, as UDTrans removes the DRAM traffic from address translation.

Runtime Overhead

In the throughput-bound performance model, memory amplification determines the relative runtime overhead. Paging with 4 KB pages introduces geometric mean runtime overheads ranging from 13% (XSBench) to 279% (PGAS), while 2 MB paging incurs 1.3% to 181% overhead. Since runtime overhead equals memory amplification by construction, Figure 4.10 expresses the same per-framework ratios as Figure 4.9, now read as throughput-bound speedup over 4 KB pages: per-node segmentation ranges from about 1.0× on VarSeg (which suffers the same 100% overhead penalty as 4 KB pages) to 3.6× on PGAS, and UDTrans ranges from 1.1× to 3.8× while eliminating translation runtime overhead (0% on every trace) since

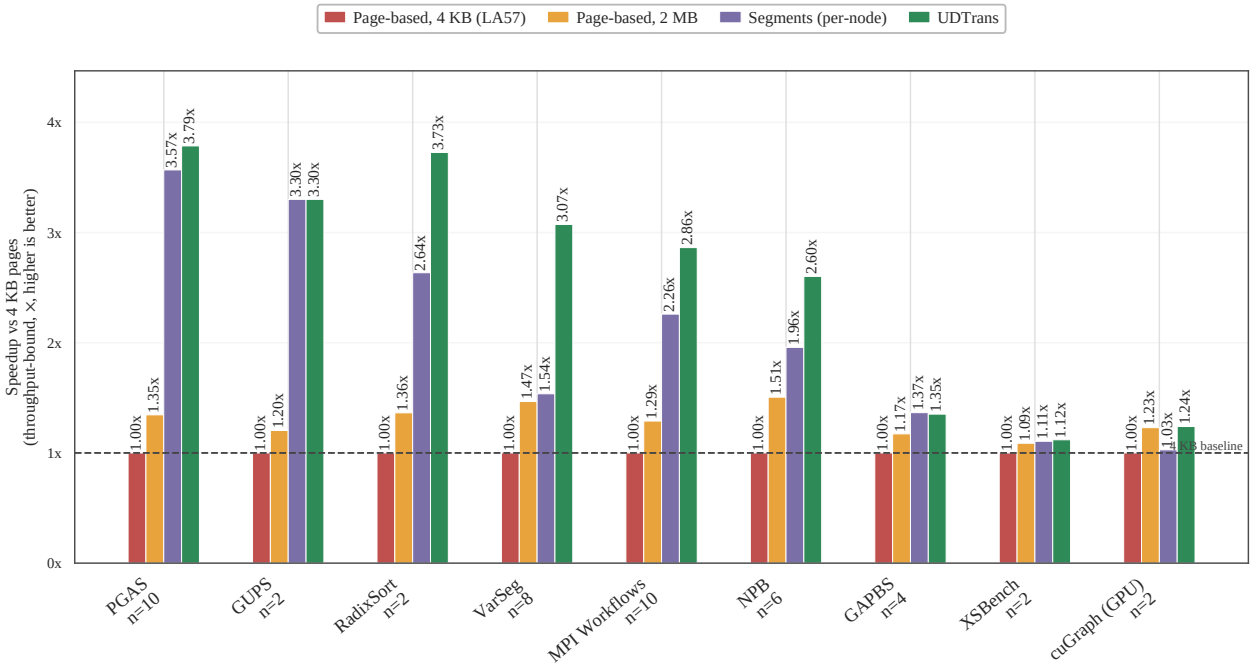


Figure 4.10: Speedup relative to 4 KB pages on the CPU/GPU traces (per-framework geometric mean of the ratio; higher is better).

every miss stays on-chip.

Power

Figure 4.11 compares each scheme's translation power for the translation hardware, expressed as a ratio to 4 KB paging on the same trace (higher is better). 4 KB paging itself ranges from 360 mW to 1.9 W across frameworks. Relative to it, UDTrans saves a geometric mean of 75× (19–116× per framework): its power stays within 16-17.4 mW on 8 frameworks, since eliminating DRAM page walks leaves only on-chip SRAM lookups, but rises to a 32.9 mW mean on NPB's IS-E bucket sort, where the same round-robin descriptor pressure that lowers its hit rate also raises SRAM lookup traffic. Per-node segmentation saves 9.4× on average (1.4-75×, dominated on the low end by cache-spilling traces such as VarSeg), while 2 MB paging saves only 1.5× (1.1-4.1×), since it still walks DRAM on every miss. These results confirm that for large-scale data footprints, DRAM translation table traversal dominates

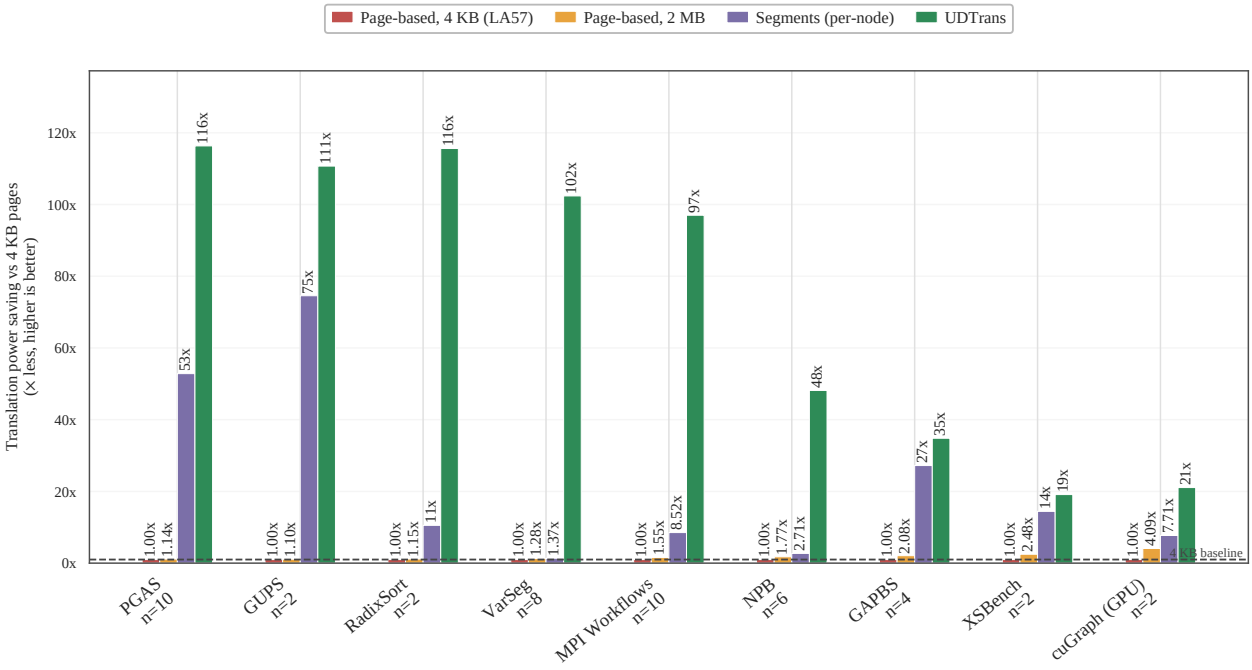


Figure 4.11: Translation power relative to 4 KB pages on the CPU/GPU traces: CACTI 22 nm array power plus page-walk DRAM traffic for a single translation unit (per-framework geometric mean of the ratio to 4 KB pages; higher is better), under an assumed 2 ns per-reference time base.

translation energy.

4.3.2 Performance on UpDown Traces

We next evaluate the four translation schemes on UpDown accelerator memory traces, with cycles reported by the simulator. The metric differs from the CPU/GPU traces in Section 4.3.1 because UpDown has a different memory hierarchy, concurrency, and memory-access granularity.

TLB Hit Rate

On the UpDown accelerator traces (Figure 4.12), all schemes achieve higher baseline hit rates because accelerator-issued DRAM references are word-level, exhibiting higher spatial and temporal locality. Nonetheless, the relative ranking across schemes remains the same.

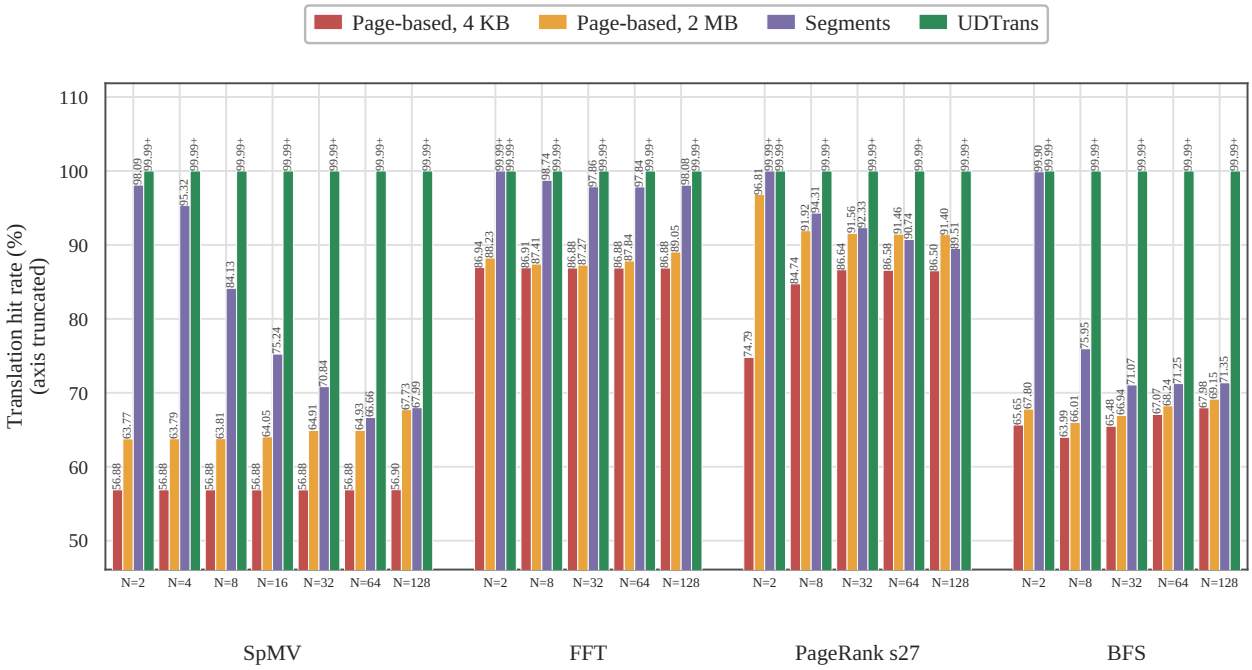


Figure 4.12: TLB hit rate on the UpDown traces. UDTrans hit rates exceeding 99.99% represent lower bounds.

The 4 KB page table achieves hit rates between 56.9% (SpMV) and 86.9% (FFT) and 2 MB huge pages improve hit rates to 63.8%–96.8%. Per-node segmentation ranges between 66.7% (SpMV) and 100.0% (FFT). The lower hit rates in the segment baseline stem from the 8-entry TLB's descriptor capacity being insufficient for descriptors that scale with the number of nodes. Finally, UDTrans incurs only compulsory misses on first access to the segment, maintaining a hit rate above 99.99% across all workloads. This result again demonstrates the advantage of keeping the translation states invariant to node scaling.

Memory Access Amplification

Misses in the TLB and Page-Walk caches for the 4 KB page introduce 24.1% to 88.8% additional DRAM accesses, peaking on irregular sparse workloads (81.8%-88.8% on SpMV and 70.0%-81.6% on BFS). Figure 4.13 plots the other three schemes' DRAM traffic with respect to the 4 KB baseline (higher is better). 2 MB huge pages reduce DRAM traffic by

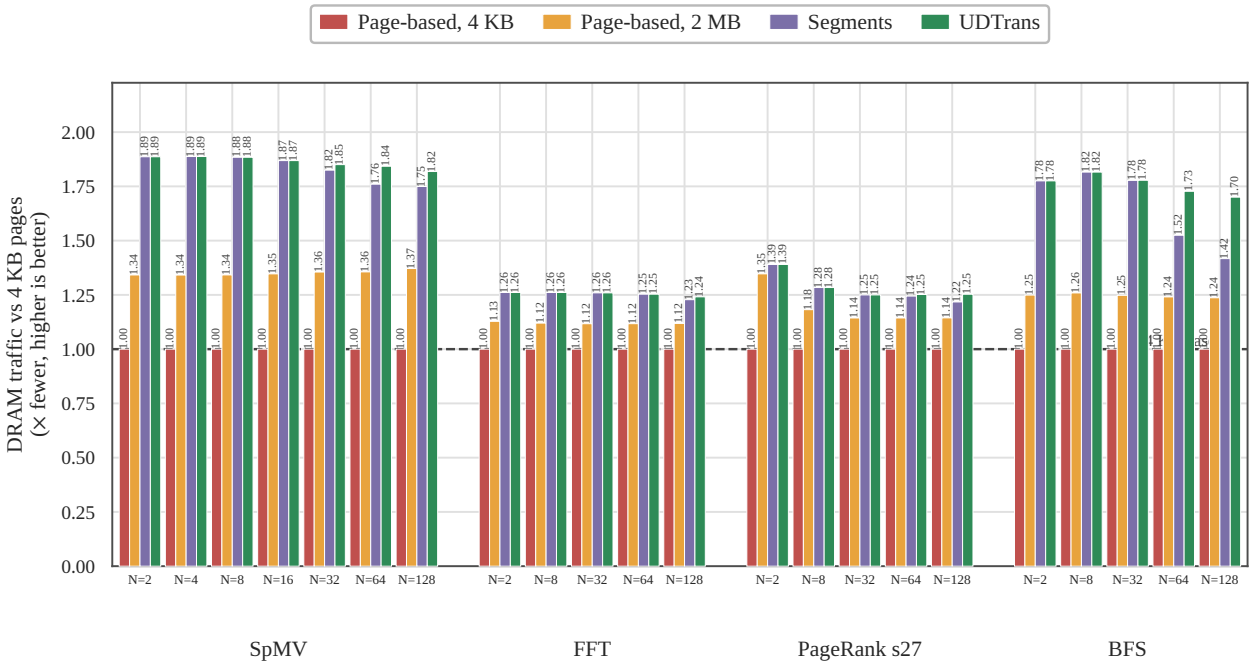


Figure 4.13: DRAM traffic relative to 4 KB pages on the UpDown corpus (ratio of total traffic; higher is better; the 4 KB baseline is the dashed 1.0× line).

only 1.12-1.37× compared to 4 KB, showing that a larger granule size cannot eliminate the TLB miss overhead from irregular access patterns. Per-node segmentation and UDTrans both avoid the overhead at low node counts (1.22-1.89× and 1.24-1.89×, respectively). At small machine scales, the segment descriptor working set fits within the segment translation cache, and memory amplification stays below 10^{-6}%. Nonetheless, as the machine scales to 64 and 128 nodes, per-node segment performance becomes worse: segmentation spills from $N = 32$ on SpMV and BFS, $N = 64$ on PageRank, and $N = 128$ on FFT, reaching 19.99% additional traffic on BFS at $N = 128$, whereas UDTrans remains zero. This highlights that conventional segments fail to express distributed data layout efficiently as the number of translation entries scales as the machine size (i.e., number of nodes).

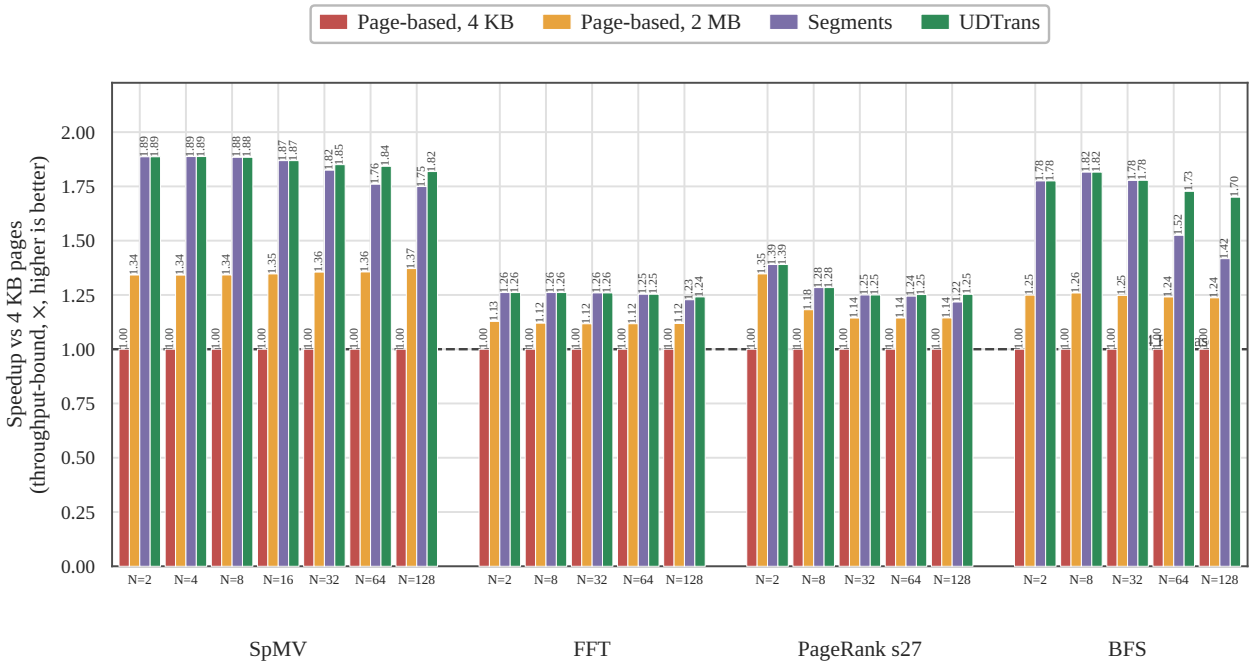


Figure 4.14: Throughput-bound speedup relative to 4 KB pages on the UpDown traces (ratio; higher is better). This ratio is analytically identical to Figure 4.13, since runtime overhead equals memory amplification by construction.

Runtime Overhead

Under high memory request concurrency, the accelerator's massive multithreading can efficiently hide translation-walk latency, so translation overhead increases with memory bandwidth consumption rather than the latency. Figure 4.14 expresses the same per-column ratios as Figure 4.13, but represents the throughput-bound speedup over 4 KB pages: 2 MB paging ranges from 1.12× to 1.37×, per-node segmentation from 1.22× to 1.89×, and UDTrans from 1.24× to 1.89×. This highlights the key architectural insight: while high-throughput architectures can effectively hide translation latency, they cannot mitigate the memory bandwidth lost to page table walks; UDTrans converts that lost bandwidth into speedup.

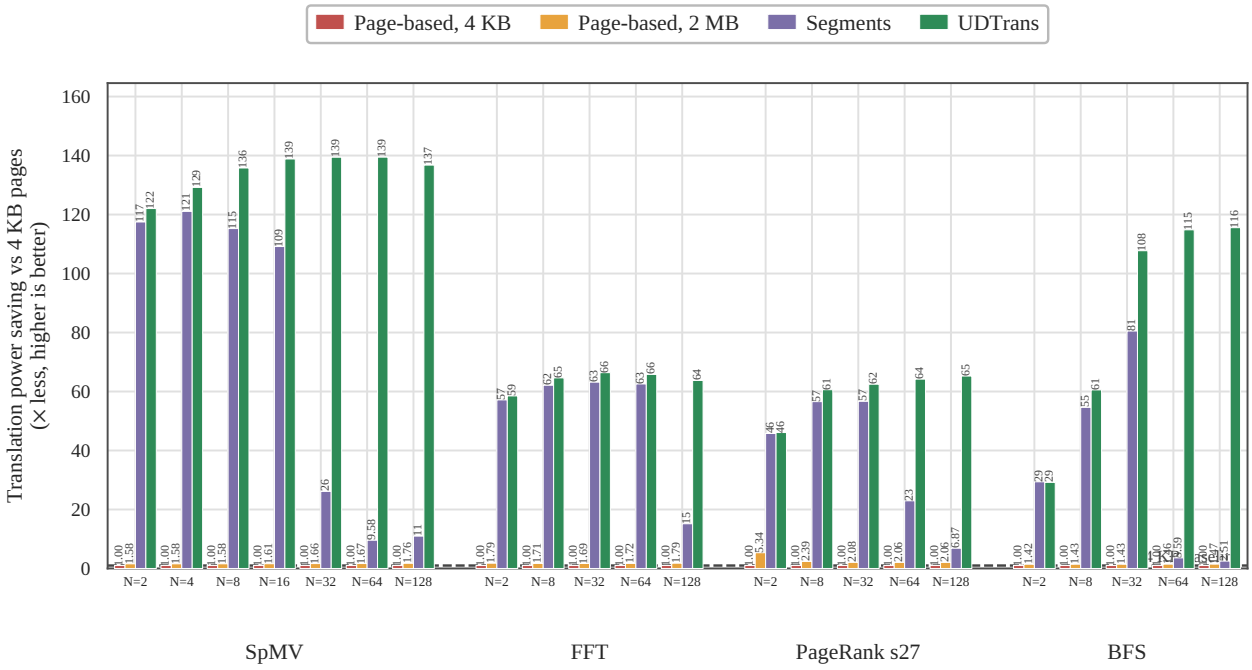


Figure 4.15: Translation power relative to 4 KB pages on the UpDown traces: CACTI 22 nm array power plus DRAM walk traffic, aggregated across all accelerator translation units over measured execution cycles (ratio to 4 KB pages; higher is better).

Power

Figure 4.15 expresses each scheme's aggregate translation power, as a ratio to 4 KB paging on the same trace (higher is better). Relative to 4 KB paging, UDTrans saves 29× to 139× (e.g., 15.2 kW vs. 108.7 W on 64 GB SpMV at $N = 64$; 2,268.9 W vs. 19.6 W on 37.3 GB BFS at $N = 128$), with approximately 97% of page-based energy expended on off-chip walk traffic. Per-node segmentation saves 2.5× to 121× and matches UDTrans only at low node counts (0.99× to 46.1× the power of UDTrans), since its descriptor working sets leave cache as N grows. 2 MB paging saves only 1.42× to 5.34×. However, the fundamental limitation of conventional segmentation worsens as node counts scale, as analyzed in the following section.

4.3.3 Locality Study: The Cost of Distributing Data

The preceding evaluations characterized translation performance under static, pre-configured data distributions. This section investigates the translation state and performance overheads required to distribute data structures across N memory nodes. In architectures featuring a multi-node, non-uniform physical address space (Section 4.1.1), a translation mechanism needs to express physical placement to retain locality control. We sweep node count N across the UpDown workloads and quantify the translation state and cache pressures required by each scheme.

Distribution State Complexity

Conventional base-and-bound descriptors specify a single contiguous range on a single node. Therefore, distributing a data structure across N nodes requires N distinct descriptors per allocation, resulting in a translation state complexity of $\mathcal{O}(N \cdot D)$ for D allocations. In contrast, UDTrans encapsulates block-cyclic distributions within hardware swizzle masks, maintaining $\mathcal{O}(D)$ state complexity (one descriptor plus a 64-bit mask per allocation) independent of N. Figure 4.16 quantifies this state scaling across application workloads: the per-node descriptor working set exceeds the 8-entry TLB capacity at small node counts and surpasses the 128-entry descriptor cache between $N = 16$ and $N = 64$. Alternatively, mapping fine-grained block-interleaved distributions with conventional segments requires thousands to millions of discrete descriptors (1.05 million in FFT rising to 1.95 million in SpMV), effectively reducing segmentation to a flat, unmanageable page table.

Translation State Costs

Figure 4.17 evaluates this state explosion across the four applications, each collected natively at every node count rather than re-split from a single trace. As N increases, the segment TLB hit rate degrades on every workload: SpMV from 98.1% (14 descriptors) to 66.7% (448

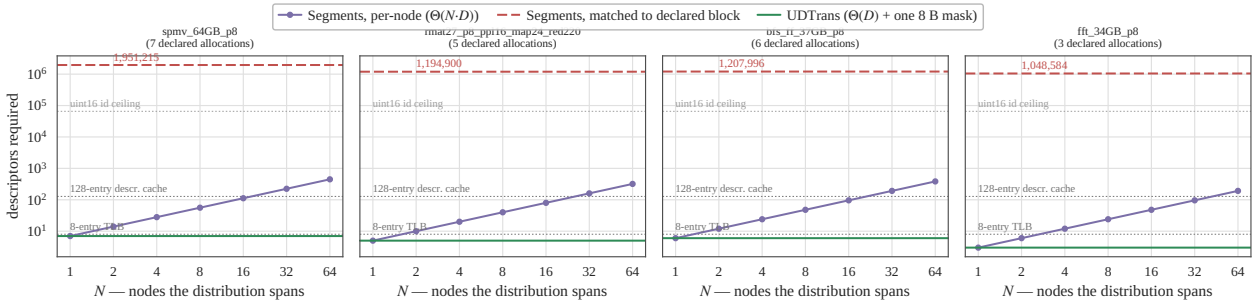


Figure 4.16: Number of descriptors required to express a N-way distribution, computed from application allocation declarations. Per-node segmentation scales as $\mathcal{O}(N \cdot D)$, exceeding TLB (8 entries) and descriptor-cache (128 entries) capacities. Unmasked fine-grained segmentation (dashed) requires 1.05 to 1.95 million descriptors, whereas UDTrans maintains constant $\mathcal{O}(D)$ state complexity.

descriptors), BFS from 99.9% (12) to 71.4% (768), PageRank from 100.0% (10) to 89.5% (640), and FFT from 100.0% (6) to 98.1% (384), whereas UDTrans maintains a 100% hit rate with only three to seven entries. As shown in the right panel, while the descriptor working set remains within the 128-entry cache, the descriptor cache successfully captures translation misses, preventing DRAM accesses beyond initial compulsory misses: SpMV stays below 8.5×10^{-7}% additional traffic through $N = 16$, its last point at 112 descriptors. However, once the working set exceeds cache capacity, descriptor misses spill to DRAM, generating 1.39% ($N = 32$), 4.70% ($N = 64$) and 3.95% ($N = 128$) additional DRAM traffic per translation on SpMV, and 13.28% ($N = 64$) rising to 19.99% ($N = 128$) on BFS—the largest translation overhead measured in this study. UDTrans incurs zero DRAM traffic regardless of N, demonstrating that it prevents cache thrashing when scaling distributed memory systems.

The measured corpus further shows that the descriptor count alone does not determine when this spill begins. FFT at $N = 64$ and BFS at $N = 32$ install an identical 192 descriptors against the same 128-entry cache, yet differ by five orders of magnitude in the DRAM traffic they generate (9.9×10^{-8}% against 0.036%); at 384 descriptors the same pairing differs by a factor of twelve (1.07% for FFT at $N = 128$ against 13.28% for BFS at $N = 64$). The

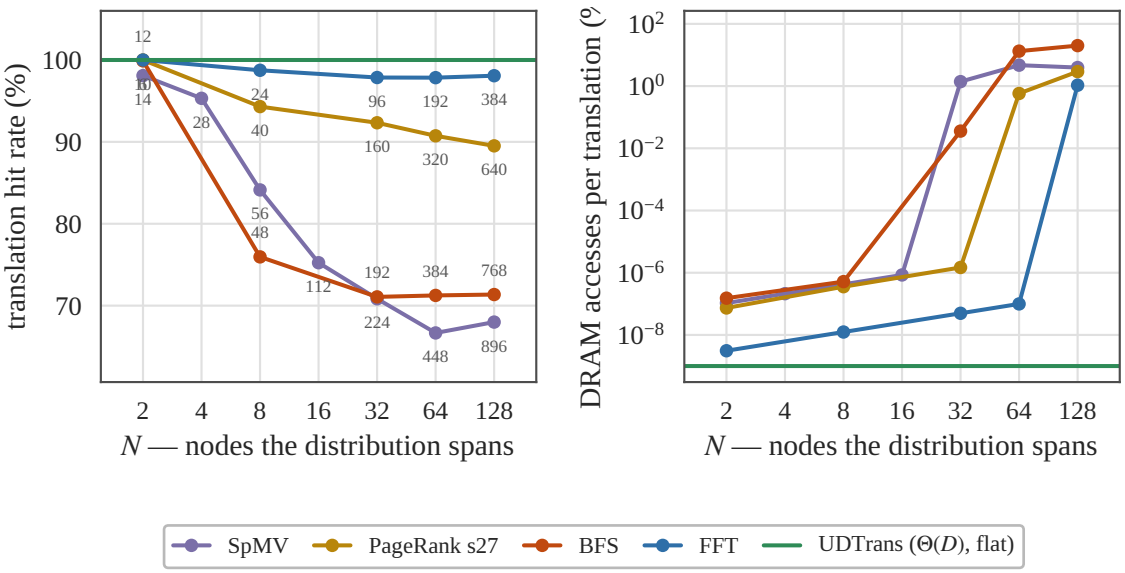


Figure 4.17: Overhead of $\mathcal{O}(N \cdot D)$ descriptors across four applications collected natively at each node count, SpMV spanning $N = 2$ to $N = 128$. Left: Segment TLB hit rate versus N (annotations indicate installed descriptors). Right: DRAM accesses per translation; the descriptor cache absorbs misses at low N on every workload and spills as the working set grows, though the crossing point depends on how many descriptors are concurrently live rather than on $N \cdot D$ alone.

operative quantity is how many descriptors are concurrently live rather than how many are resident: FFT sweeps its arrays sequentially and holds near-zero cost across a 64-fold growth in descriptor population, from 6 descriptors at $N = 2$ to 384 at $N = 128$, before crossing in a single step, whereas BFS's frontier expansion touches many partitions within a level and pays the most per descriptor.

Load Balance

Figure 4.18 evaluates the load balancing efficiency enabled by data distribution. Mapping each allocation to a single conventional segment descriptor confines the entire data structure to a single node, resulting in an extreme load imbalance factor of N (64× the mean load at $N = 64$). While splitting allocations into per-node segment descriptors restores load balance (achieving a peak-to-mean load ratio of 1.07 at $N = 64$), it incurs the substantial

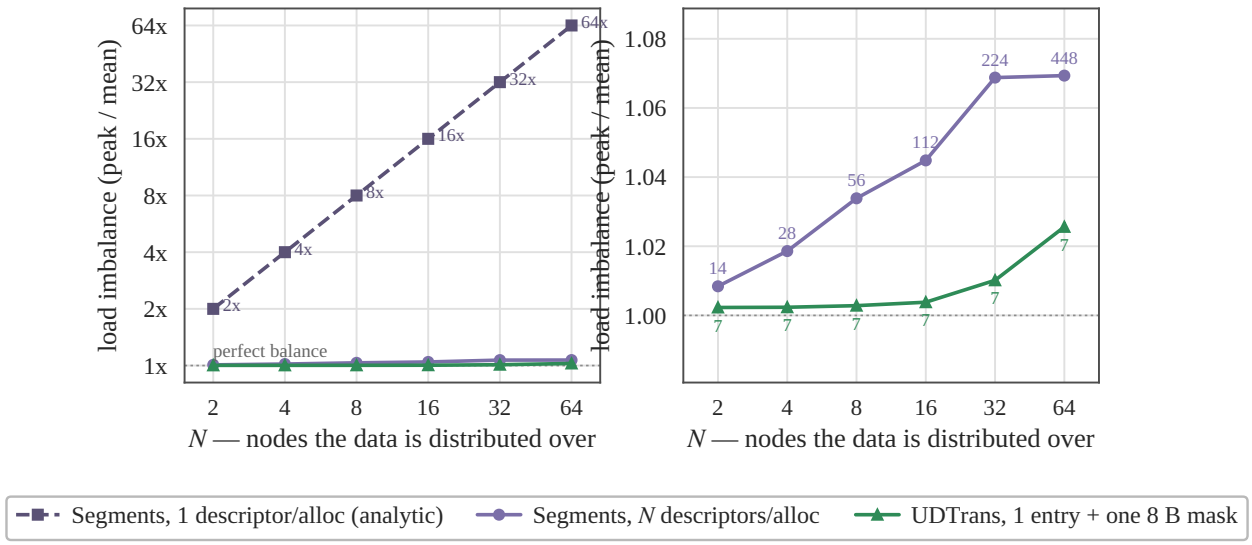


Figure 4.18: Load balance versus node count on the shared SpMV trace prefix. Left: Monolithic single-descriptor allocations cannot span nodes, yielding an imbalance factor of N. Right: Per-node segmentation and UDTrans achieve balanced distributions, but with contrasting descriptor requirements: 448 descriptors versus 7 descriptors at $N = 64$.

translation overhead of 448 active descriptors. UDTrans matches this load distribution quality (peak-to-mean ratio of 1.03 at $N = 64$) using only seven translation descriptors and their respective 64-bit swizzle masks—achieving equivalent placement efficacy with nearly two orders of magnitude less translation state.

Conditions on Swizzle Masks

We note that swizzle mask configuration is an explicit, software-guided abstraction with specific structural constraints. The target physical distribution needs to map to power-of-two block-cyclic layouts to conform to the 64-bit mask encoding (consistent with the stride alignment discussed in Section 4.2.3). Furthermore, sub-optimal mask specifications can degrade locality if mismatched with application access patterns. However, modern parallel and PGAS runtime environments natively maintain the required distribution metadata: UPC block distributions, Chapel domain maps, and OpenSHMEM processing element mappings expose the block sizes and node cardinalities needed to configure UDTrans descriptors [7,

21, 15, 10].

4.4 The Trade-off Between Translation State and Placement Flexibility

Our findings demonstrate the importance of decoupling address-translation granularity from physical data placement granularity. In conventional page-based systems, translation reach and placement granularity are tightly coupled to the page size. Our results illustrate the consequences of this coupling: expanding page sizes from 4 KB to 2 MB increases TLB reach and improves hit rates only for workloads with structured spatial locality, while still incurring up to 181% runtime overhead from page walks in irregular workloads. Furthermore, huge pages enforce coarse-grained placement, rendering fine-grained distributed data placement impossible because the page serves as the indivisible unit of allocation and migration. Conversely, while flat segmentation offers unbounded reach per descriptor, scaling distributed allocations across N nodes inflates translation state proportionally ($\mathcal{O}(N \cdot D)$), causing TLB thrashing and generating substantial DRAM miss traffic once descriptor working sets exceed cache capacity.

UDTrans resolves this dilemma by decoupling reach from placement: arbitrarily sized segment descriptors provide unbounded reach, while the block size defined by the 64-bit swizzle masks controls the distribution granularity. Therefore, UDTrans achieves near-optimal load balance across distributed nodes while reducing translation state by up to two orders of magnitude, eliminating DRAM amplification, and minimizing translation power to on-chip array access levels.

4.5 Summary

UDTrans presents an address translation scheme that decouples translation reach from physical data placement by pairing variable-sized segment descriptors with 64-bit swizzle masks. Across a comprehensive 46-workload CPU/GPU traces spanning 64 GB to 565 GB footprints, standard 4 KB paging suffers from severe TLB thrashing ($<$1% hit rate on 28 workloads) and inflates memory traffic by 164.8% on average. In contrast, UDTrans achieves $>$99.99% hit rates across 33 workloads (and $\geq$85.6% in all cases) with zero DRAM amplification and a geomean of 75$\times$ reduction in translation power. On native UpDown accelerator workloads with measured cycle times, conventional page walks consume 24%–89% of total memory bandwidth and up to 139$\times$ more power than UDTrans: penalties that latency-tolerant execution models cannot hide. Furthermore, our locality scaling study confirms that distributing data structures across N nodes scales the descriptor working set of flat segmentation as $\mathcal{O}(N \cdot D)$, degrading hit rates to 66.7% and incurring off-chip cache spills, whereas UDTrans maintains constant $\mathcal{O}(D)$ descriptor state while matching optimal load distribution. As distributed and near-memory architectures scale to petabyte-class memories, decoupling translation reach from placement precision provides a robust and scalable foundation for next-generation parallel memory systems.

CHAPTER 5

GLMALLOC: HIERARCHICAL MEMORY MANAGEMENT FOR DISTRIBUTED MEMORY ARCHITECTURES

Chapter 4 established that the UDTrans hardware translation architecture achieves single-cycle address translation and minimal translation metadata overhead by mapping memory through coarse-grained swizzle segments. However, UDTrans relies on an architectural premise: that software allocates memory in a few large, globally accessible segments and expresses data distribution across physical nodes via swizzle bitmasks rather than through thousands of discrete page translation descriptors.

While this coarse-grained model suits static, monolithic data structures, modern scale-out data-intensive workloads, ranging from irregular graph analytics to paged Key-Value (KV) cache management in large language model (LLM) serving, routinely exhibit fine-grained, dynamic, data-dependent memory allocation patterns. This discrepancy creates a fundamental architectural dilemma for distributed accelerator architectures:

1. **Translation State Explosion**: If the software runtime provisions a distinct hardware translation descriptor for every fine-grained allocation, the hardware translation table would soon run out of capacity, triggering TLB thrashing and degrading performance.

2. **Centralized Arbitration Bottleneck**: If the runtime services millions of concurrent, fine-grained allocation requests through a traditional centralized allocator (e.g., relying on global heaps, locks, atomic counters, or centralized managers; Section 5.3.3), synchronization traffic, network serialization, and lock contention collapse system scalability.

To resolve this dilemma, this chapter presents **GLMalloc**, a two-level hierarchical memory management control plane co-designed with the UDTrans hardware translation architecture. A central thesis of this work is that UDTrans's coarse-grained segment requirement

is not an operational limitation imposed on applications, but rather an architectural invariant produced by GLMalloc. GLMalloc reconciles fine-grained, highly concurrent application allocations with coarse-grained hardware segment translation by separating memory management along privilege, granularity, and locality boundaries:

- **Level-One (Privileged, Global, Coarse-Grained)**: A privileged global segment manager (`DRAMalloc`) allocates large swizzle segments spanning multiple memory nodes in a 2-dimensional physical address space, configuring hardware swizzle bitmasks, base offsets, and memory protection bounds.

- **Level-Two (Unprivileged, Decentralized, Fine-Grained)**: An unprivileged, user-level library (`MemoryPool`) partitions each segment into private, per-lane sub-pools managed via single-level bitmaps with local scratchpad SRAM caching. Each execution lane services its own allocations and deallocations locally from private metadata, eliminating atomic synchronization, locking, and hardware translation modifications on the critical path.

By enforcing this separation, privilege changes hands once at segment granularity: the operational cost of privileged, centralized coordination is amortized over millions of subsequent unprivileged local allocations. Furthermore, to mitigate spatial demand skew across local allocators without reintroducing centralized managers, GLMalloc incorporates an unprivileged, three-tier load-balancing mechanism: fast intra-device rotated-origin rerouting, inter-device region redistribution, and backed up by inter-node rerouting as the final recovery tier.

5.1 Design Principles

GLMalloc separates permission management from runtime memory management as illustrated in Figure 5.1. The Level-One is managed by a privileged global manager and deter-

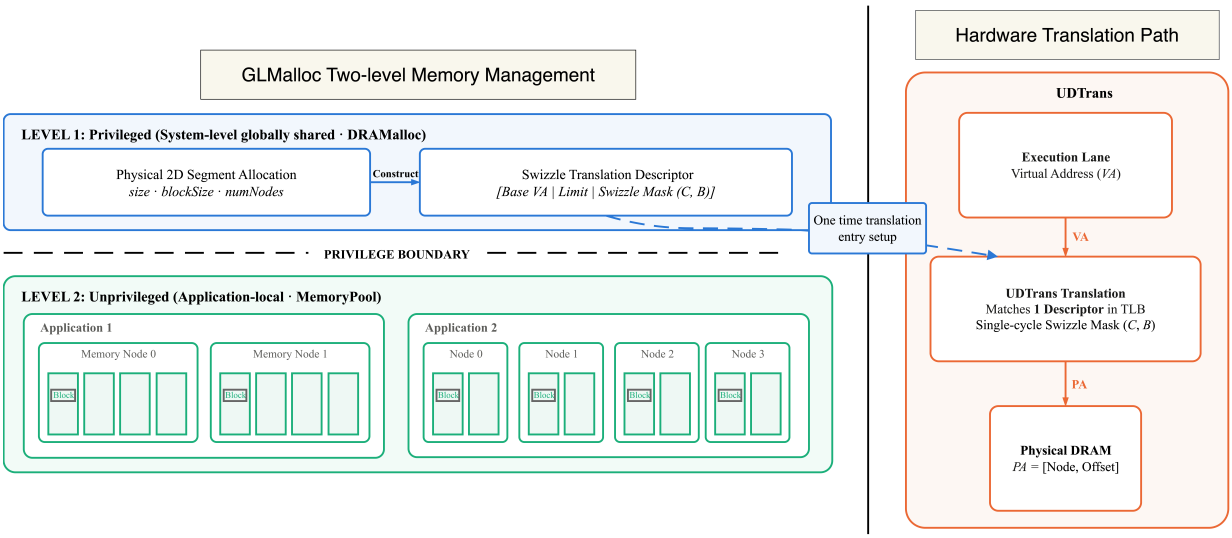


Figure 5.1: The two-level memory management design and the translation path upon a memory access.

mines the physical memory accessible to each application process. The second level is executed by an unprivileged, application-level library that determines fine-grained word-level placement within the physical allocation.

Upon a memory access instruction, the address is validated and translated via the swizzle translation entry set up by the first-level privileged manager, as shown in the right half of Figure 5.1.

5.1.1 Level-One: Privileged Global Allocation

The Level-One privileged global manager, called `DRAMalloc`, takes the 2-dimensional physical address space defined in Chapter 4 as input and allocates swizzle segments to application processes. An allocation request specifies both a segment's aggregate capacity and its spatial distribution across a range of memory nodes. The allocation unit is a **swizzle segment**, a global virtual address range distributed in a block-cyclic physical layout across a consecutive range of memory nodes.

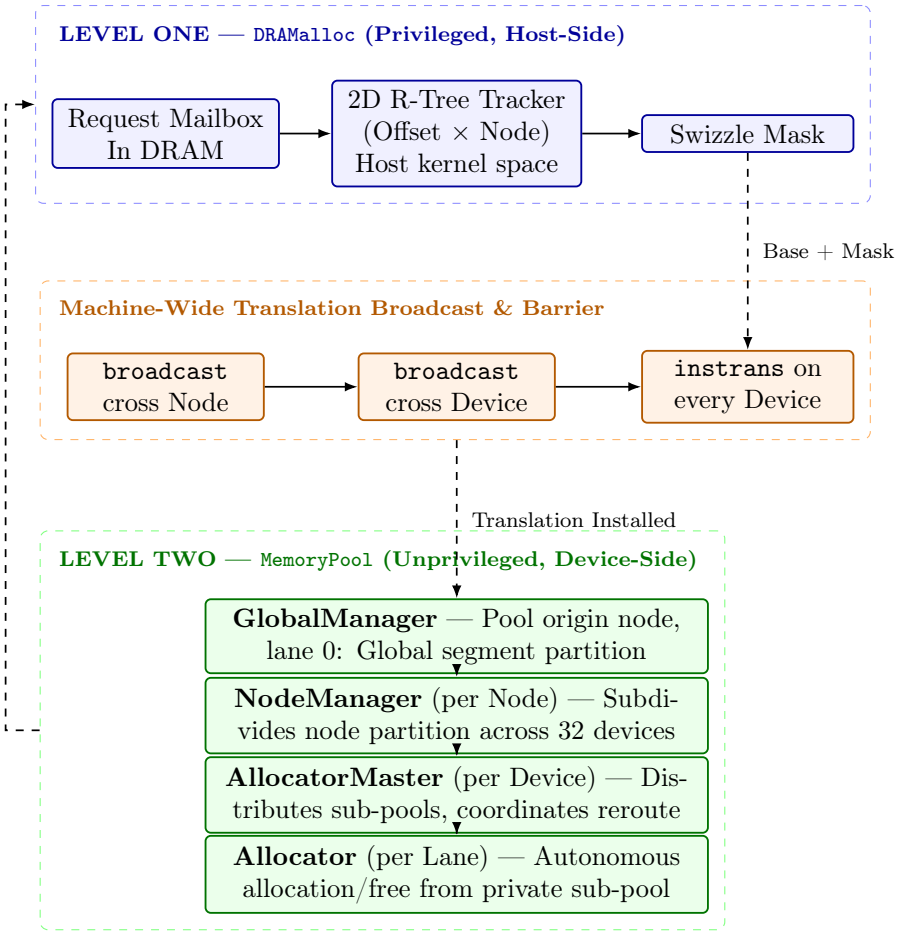


Figure 5.2: The two-level memory management scheme of GLMalloc. Privileged Level-One allocation occurs once per major data structure and broadcasts translation descriptors to accelerator devices. Subsequent fine-grained allocations are served by unprivileged distributed Level-Two.

The host-side interface provides the principal allocation primitive:

$$\texttt{mm_malloc_global}(\text{size}, \text{blockSize}, \text{numNodes}, \text{startNode}) \tag{5.1}$$

where blockSize represents the striping granularity, numNodes denotes the number of participating physical memory nodes, and startNode designates the physical origin node. Device lanes invoke the function on the host asynchronously via a memory-mapped region in DRAM, passing the allocation parameters along with an event continuation handle.

`DRAMalloc` executes on a dedicated, privileged host processor. Upon processing an allo-

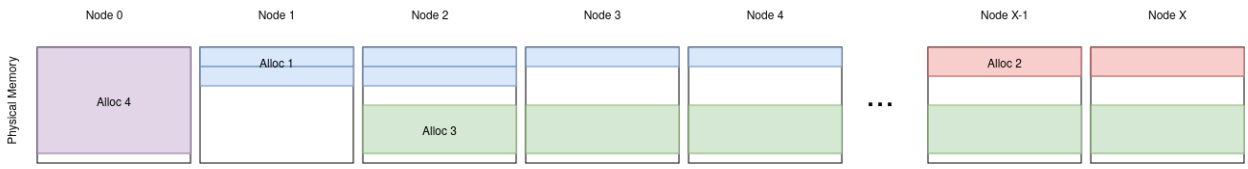


Figure 5.3: `DRAMalloc` 2-dimensional allocation with R-tree.

cation request, `DRAMalloc` performs the following sequence:

1. **Extent Allocation**: Allocates free physical memory using a 2-dimensional R-tree structure defined over the (offset × node) 2-D space (Figure 5.3).

2. **Swizzle Mask Construction**: Derives the hardware swizzle bit-mask parameters $C = \log_2(\text{blockSize})$ and $B = \log_2(\text{numNodes})$.

3. **Machine-Wide Broadcast**: Issues a hierarchical broadcast across the system to all target nodes to install the translation entry on all translation tables.

4. **Barrier and Notification**: Upon receiving the acknowledgment, the manager signals the caller's continuation with the allocated virtual address base.

Amortization of Privileged Overhead

Because Level-One segment allocation requires global synchronization across a distributed system, its latency and cost are significant, so we restrict Level-One allocation to one per aggregate data structure, and each application program needs only a few throughout execution. A single Level-One segment may then yield hundreds of thousands of user-level fine-grained allocations across nodes. Therefore, the overhead of the privileged global coordination is amortized by application runtime and the orders-of-magnitude dynamic allocation and deallocation operations executed entirely at user level.

Hardware Swizzle Mask Encoding and Locality Control

The Level-One allocation parameters determine how data is distributed across nodes. With $C = \log_2(\text{blockSize})$ and $B = \log_2(\text{numNodes})$, the hardware swizzle logic decomposes an incoming segment offset into an intra-block offset (C bits), a node selection index (B bits), and a block index. Formally, virtual block index i maps to physical node:

$$\text{Node}(i) = \text{startNode} + (i \bmod \text{numNodes}) \tag{5.2}$$

at intra-node byte offset:

$$\text{Offset}(i) = \lfloor i/\text{numNodes} \rfloor \cdot \text{blockSize} \tag{5.3}$$

When numNodes $= 1$, the striping logic falls back to an identity mapping, resulting in node-local allocation.

With the swizzle mask, application programs can configure the physical data distribution across nodes by passing parameters to the Level-One allocator. When blockSize is set to default interleaving, data is striped across memory nodes. Conversely, when runtime software sets blockSize equal to the per-node quota (like how MPI programs partition their data), the swizzle logic maps contiguous block ranges entirely to the local node. Under this configuration, each node's physical slice corresponds to the aggregate sub-pools of that node's lanes, converting global memory operations into node-local accesses. Figure 5.4 demonstrates how this single Level-One parameter flexibly controls whether lower-level allocations reside locally or remotely, eliminating cross-node network traffic on memory operations.

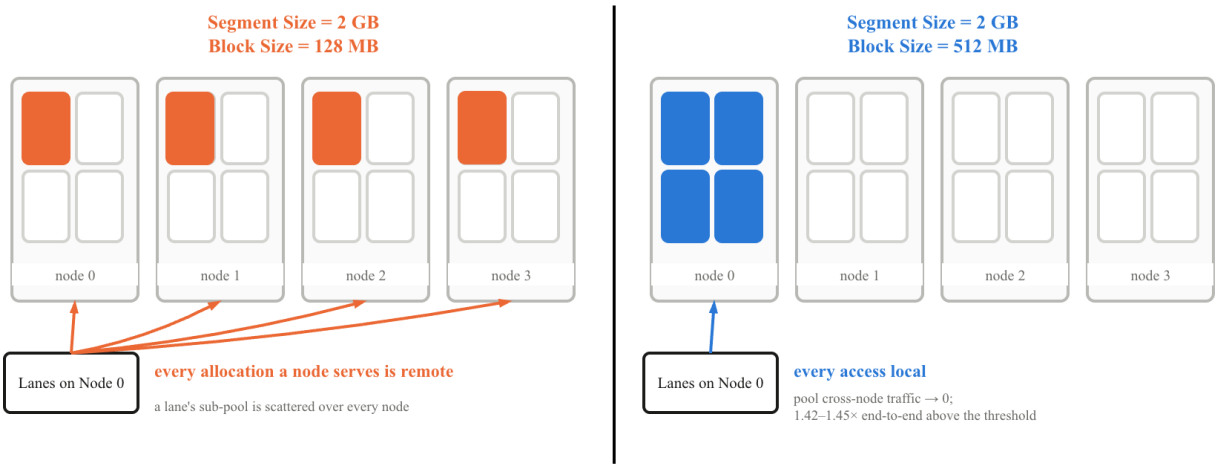


Figure 5.4: One parameter decides whether the lower level is local or remote.

Process Isolation Enforcement

The hardware translation mechanism in Chapter 4 and hardware process isolation boundaries enforce process isolation. The Level-One global manager maintains each application's allocation in a 2D R-tree and translation shadow tables, which are privileged and inaccessible to user code, and it installs translation entries in privilege mode. Hardware translation entries carry base, limit, physical origin, and permission bits, and the translation unit validates every memory access against these bounds. A Level-Two allocator manipulates only user-level bitmap words inside a segment assigned by Level-One; no sequence of allocations, frees, reroutes, or region migrations can produce an address that translates outside the segment bounds. Sub-pool reassignments transfer ownership of capacity already inside the segment, and all the allocation blocks in Level-Two are contained within Level-One. Process isolation is guaranteed at segment granularity between different applications sharing global memory, leaving unprivileged user programs to allocate and balance memory within their authorized space.

5.1.2 Level-Two: Unprivileged, Decentralized Fine-Grained Allocation

Level-Two, implemented by the user-level `MemoryPool` library, operates within a pre-allocated Level-One segment to enforce isolation. Its allocation granularity is an application-specific **block**. To eliminate global synchronization bottlenecks, `MemoryPool` employs a hierarchical partitioning scheme that subdivides the global segment along architectural boundaries (node, device, and then lane; see Section 5.2.1):

- **GlobalManager**: Receives the segment allocation from Level-One DRAMalloc and partitions the address space across participating nodes.
- **NodeManager**: Subdivides the node's local partition evenly across its constituent devices.
- **AllocatorMaster**: Assigns dedicated sub-pools to lanes within each device and coordinates intra-device load balancing.
- **Allocator**: An allocation engine running on each lane that services application requests directly from its private sub-pool.

Per-Lane Partitioning of Memory Pool

By partitioning the memory pool into private sub-pools per lane, each lane maintains exclusive ownership over its local allocator state. As a result, a lane allocator accesses only local metadata for allocations and deallocations, eliminating expensive shared free lists, spinlocks, and global managers on the critical allocation path. This decentralized design eliminates inter-lane contention and provides scalable parallel throughput.

Bitmap and Scratchpad Caching

Each sub-pool's allocation state (free versus allocated for every block) is represented as a bitmap stored in local node DRAM. To avoid high-latency DRAM round trips on the critical allocation path, the allocator uses the lane's private 64 KB scratchpad SRAM as a fast metadata cache holding the frequently accessed, active bitmap working set. The cache size is runtime configurable.

Each lane exclusively caches and manipulates its own sub-pool bitmap words within its private scratchpad SRAM bank to avoid sharing metadata across lanes on the allocation fast path. On the common-case fast allocation path where active bitmap words hit in the scratchpad cache, the allocator serves the request entirely from local on-chip SRAM with zero DRAM loads and deterministic latency. Figure 5.5 illustrates this metadata management hierarchy and the residency models.

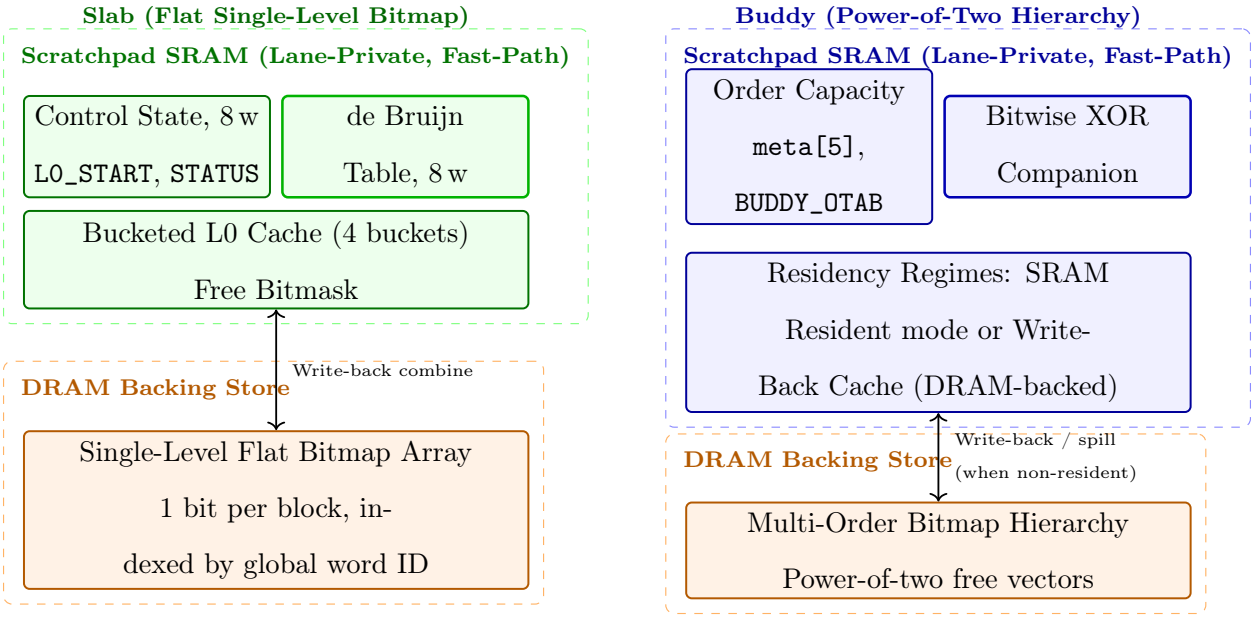


Figure 5.5: Level-Two metadata caching hierarchy across the two allocator backends.

Allocator Backends and Backend-Specific Data Structures

Because Level-Two is decoupled from the privileged Level-One global manager, applications can select allocator backends tailored to their allocation and access patterns at compile time. The backends' metadata structure differs as follows:

1. **Fixed-Size Slab Backend**: Optimized for uniform-size allocations, minimizing internal fragmentation. Its data structures comprise:

    - **Flat Single-Level Bitmap**: Tracks block availability with one bit per block arranged in 64-bit words.
    - **Bucketed Scratchpad Cache**: Statically partitions the bitmap words across a runtime-configurable number of scratchpad cache buckets. Each bucket maintains a designated contiguous range of bitmap words, local head and tail indices, and a per-bucket availability bitmask, refilling from DRAM upon exhaustion.
    - **Scratchpad de Bruijn Table**: A 64-entry constant multiplication table that performs constant-time ($O(1)$) lowest-set-bit identification without conditional branching.

2. **Multi-Order Buddy Vector Backend**: Provides variable-size power-of-two block allocation for workloads with dynamic request sizes. Its non-shared data structures comprise:

    - **Multi-Order Bitmap Table** : An array indexing per-order free-block bitmaps spanning orders $0 \leq k \leq \text{max_order}$.[1]

1. **Scratchpad-Resident Mode for Buddy**: The *Scratchpad Buddy* is the resident case where the order bitmap fits within the lane's scratchpad SRAM cache budget. In this mode, all per-order bitmaps remain in scratchpad SRAM, so allocations and coalescing do not require DRAM loads/stores. When sub-pools exceed scratchpad SRAM capacity, the buddy allocator operates in DRAM-backed mode via a direct-mapped write-back scratchpad cache that dynamically spills and refills bitmap words.

- **Bitwise Companion Indexing**: Companion buddy block addresses are computed in a single execution cycle via bitwise XOR (`block_idx ^ (1 << order)`), eliminating table lookups and pointer chasing during coalescing and splitting.
- **Order-Capacity Guards and Bit-Scanning**: An aggregate free-block counter gates order traversals to skip empty orders, followed by a shift-and-test loop to locate available blocks within a bitmap word.

5.1.3 Level-Two: Hierarchical Load Balancing

While static per-lane partitioning maximizes fast-path throughput, severe spatial skew in application demand can cause individual lanes to exhaust their sub-pools while neighboring lanes retain abundant free capacity. GLMalloc addresses load imbalance through an unprivileged three-tier hierarchical mechanism: intra-device rotated rerouting and inter-device region redistribution backed by inter-node rerouting.

Tier 1: Intra-Device Rerouting

When a lane allocator exhausts its local sub-pool, it first attempts a bounded number of local retries to wait for pending in-flight bucket refills. If the local sub-pool remains empty, the lane triggers intra-device rerouting.

Because all lanes within an accelerator device share a common scratchpad address space, the overflowing lane can query the device-level availability state to select a candidate lane with available blocks for rerouting. The device maintains two shared availability bitmasks in scratchpad memory: a lane-availability vector indicating lanes with free blocks, and a cache-availability vector indicating lanes whose free blocks are currently cached. The lane prioritizes free blocks already cached to maximize fast-path requests.

To prevent multiple overflowing lanes from colliding on the same rerouting candidate, each lane starts its search from a rotated origin equal to its lane ID, masking out its own

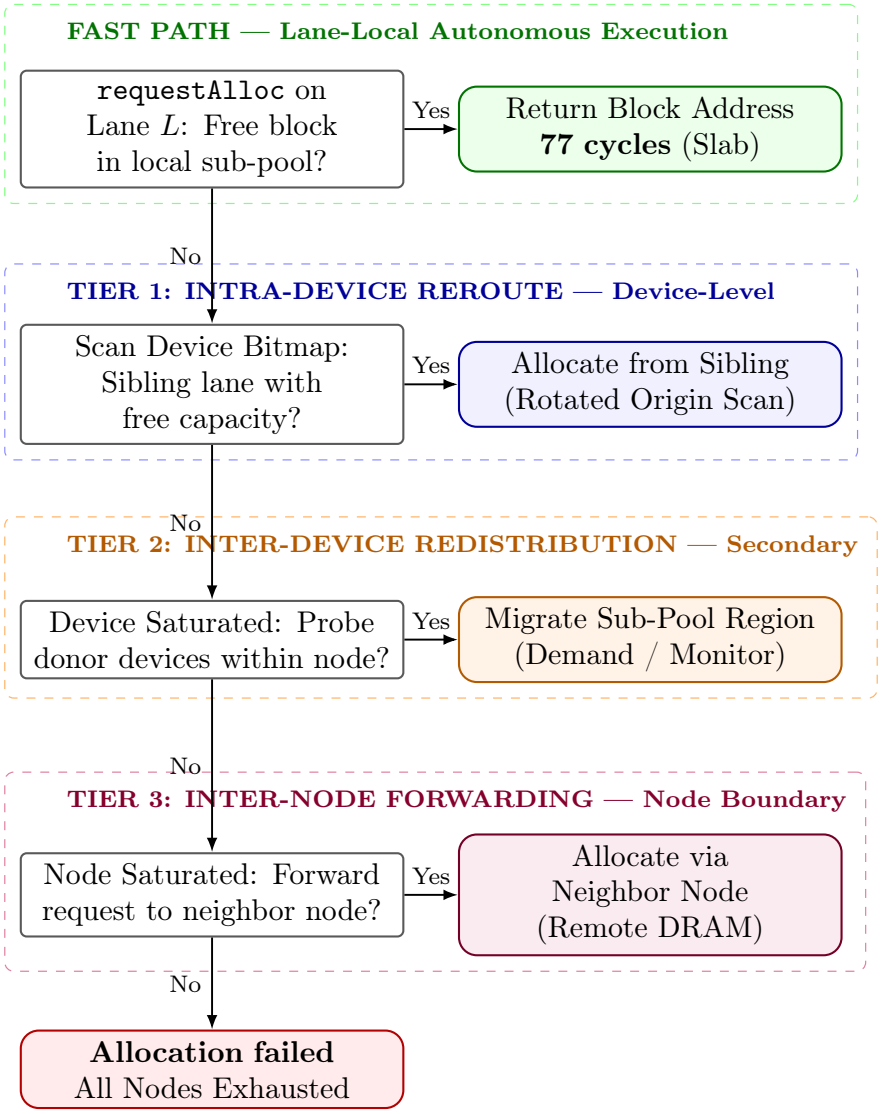


Figure 5.6: Three-tier hierarchical load balancing in GLMalloc.

index. The lane examines the shared vector and directs its allocation request directly to the selected sibling lane. If a selected sibling has concurrently exhausted its capacity, it forwards the request to the next available candidate. State transitions in the shared availability vectors are updated atomically via cross-lane compare-and-swap (`CAS`) operations. Intra-device rerouting is contained within the device's scratchpad memory, executing with low latency and zero inter-device communication.

Deallocations preserve sub-pool ownership: each sub-pool encodes its current lane owner ID. When a deallocation request reaches the allocator, it checks whether the sub-pool containing the block is owned by the same lane. Otherwise, it reroutes the request to the owner lane.

Tier 2: Inter-Device Sub-Pool Redistribution

When spatial demand skew concentrates work onto an entire device such that intra-device rerouting cannot satisfy requests, GLMalloc activates inter-device redistribution. Sub-pool redistribution transfers the ownership of sub-pool capacity across devices within the same node:

- **Demand-Driven Region Migration**: Triggered synchronously upon allocation failure when all lanes on a device run out of free blocks. The saturated device probes candidate donor lanes on other devices within the node. A probed donor lane consents to migration only if it can donate a substantial portion of its capacity, ensuring meaningful capacity recovery due to the inter-device communication overhead. Upon donor consent, both donor and recipient flush their metadata to DRAM, and ownership of the donor's sub-pool region transfers to the recipient. Specifically, ownership travels with the region: subsequent block deallocations are routed to whichever lane currently owns the region at release time, preventing dangling pointers or cross-lane ownership livelocks.

- **Asynchronous Usage Monitoring**: An asynchronous monitoring daemon running in the background periodically gathers device-level utilization telemetry across execution phases. Applications can optionally configure periodic poll-driven region swaps as an opt-in rebalancing mode for predictable phase-based workloads.

Tier 3: Inter-Node Request Forwarding at the Node-Level Boundary

When spatial demand skew exceeds the boundaries of an individual physical node (i.e., all 32 devices on a node saturate while neighboring nodes retain unutilized physical memory), load balancing reaches the cross-node boundary. In GLMalloc, physical sub-pool placement is fixed by the Level-One swizzle translation mask during privileged Level-One initialization to preserve allocation locality. Therefore, we avoid cross-node sub-pool migration, which would break the locality assumption.

To overcome this constraint without centralized coordination, Tier 3 forwards allocation requests to neighboring nodes. When an execution lane exhausts both its local sub-pool (fast path) and all donor devices on its local node (Tier 2), it constructs a remote allocation event forwarded to the same $(\text{device}, \text{lane})$ coordinate on a neighbor node in ring order $((\text{node} + \text{hop}) \bmod N)$. Preserving the lane position distributes the overflowing node's lanes across the recipient node's devices, preventing hot spots. Upon receiving a forwarded request, the remote node attempts to fulfill it from its own sub-pool or local Tier 1/2 rebalancing. To prevent unbounded allocation loops, forwarded requests are bounded to a single hop and cannot be re-forwarded. If it still fails, the user decides whether to retry with a different node.

For deallocations, each block header encodes its absolute physical base address, allowing the deallocating lane to return blocks to the owning lane via address arithmetic and ownership rerouting similar to Tier 2.

5.2 Methodology

5.2.1 Simulation Environment and Machine Model

We conduct all experiments on the cycle-accurate UpDown architecture simulator [102, 100, 101, 25]. The baseline system model reflects a massively parallel distributed accelerator organized into hierarchical clusters:

- **Device**: Each device contains 64 execution **Lanes** and private scratchpad SRAM (64 KB total, accessible in 1-4 cycles).
- **Node**: Each node comprises 32 devices, totaling 2,048 lanes and local DRAM controllers.
- **Memory Latency Hierarchy**: Local DRAM access incurs a baseline latency of 200 cycles, while remote inter-node DRAM access requires 1,300 cycles (a 6.5× NUMA latency ratio). The lanes operate without hardware data caches, so all metadata memory accesses directly reach DRAM.

5.2.2 Evaluation Metrics

The rest of the chapter evaluates GLMalloc with four metrics:

1. **Throughput**: Aggregate allocation operations completed per 1,000 cycles across lanes (or allocations per cycle).
2. **Latency**: Individual operation cost in processor cycles, measured with different contention levels.
3. **Delivered Capacity Ratio**: The ratio of successfully served allocations to total

offered requests, evaluated against the theoretical analytic capacity ceiling:

$$\text{Capacity Ratio} = \frac{\text{Served Requests}}{\min(\text{Offered Demand}, \text{Physical Capacity})} \tag{5.4}$$

4. **Per-Lane Normalized Cycles**: Cycles per operation per lane. This metric indicates hidden centralized bottlenecks: any shared arbitration or lock contention scales super-linearly with lane count.

For production allocator comparison in Section 5.3.4, we evaluate on microarchitecturally invariant metrics: instruction-retired density, memory-hierarchy penetration, and capacity delivery.

Notation

The evaluation also uses two notations for skew and occupancy that can be computed from an application's data.

- **Skew** σ is the ratio of the busiest allocator's demand to the mean. [2] We use σ_{lane} to denote the standard deviation over the machine's lanes, and σ_{dev} over a node's devices.
- **Occupancy** ρ is offered demand divided by total sub-pool capacity. A lane's demand exceeds its private sub-pool once $\sigma_{\text{lane}} > 1/\rho$. In data-driven graph applications, out-degrees are drawn from a Zipf distribution with weights $w_i = (i+1)^{-\theta}$.

5.2.3 Workloads and Evaluation Configurations

The evaluation consists of five workloads: synthetic workloads, irregular pipeline processing, data-driven graph workloads, and real-world transformer inference (Table 5.1).

2. σ changes as the lane count varies: on one R-MAT scale-14 ingest, σ_{lane} rises from 5.1 at 256 lanes to 33.4 at 2,048 as the same degree mass is spread over more lanes.

Table 5.1: Evaluation workloads and dataset parameters spanning microbenchmarks and four distributed accelerator application pipelines.

Tier	Machine Size	Pool Range	Operations
`synthetic`	2-128 nodes, 2,048-131,072 lanes	1-64 nodes	32,768-2,097,152
`frontier_msr`	1-32 nodes, 2,048-65,536 lanes	16 MB-64 GB per node	8,192-262,144
`graph_ingest`	1-8 nodes, 2,048-16,384 lanes	16 MB-2 GB; 128 B and 4 KB	Up to 7.0×10^6
`bfs_dynarray`	1-32 nodes, 2,048-65,536 lanes	64 MB (skew); 64 GB (feasibility)	Up to 3.4×10^7
`llm_serve`	2-64 nodes, 4,096-131,072 lanes	0.5-128 GB	Up to 9.4×10^8

- **Synthetic Microbenchmarks (`synthetic`)**: Examines the core allocator operations under different concurrency windows, spatial placement, memory footprints, and skew distributions.

- **Frontier (`frontier_msr`)**: An irregular frontier insertion application with decoupled compute and allocator lanes to evaluate footprint scalability, scratchpad residency, and inter-device recovery.

- **Graph Ingestion (`graph_ingest`)**: Dynamically construct a neighbor-list-format graph from an edge list to examine load balancing under data-driven power-law degree skew, where vertex degree determines the per-lane allocation volume.

- **Dynamic-Frontier BFS (`bfs_dynarray`)**: A conventional data-dependent, memory-intensive workload that synchronizes on global barriers. Degree skew stresses the memory allocator's load balancing. Frontier-size variation across iterations also highlights the necessity of dynamic allocation versus static worst-case reservation.

- **Paged-Attention LLM Serving (`llm_serve`)**: A high-concurrency transformer KV-

cache serving engine managing dynamic request lengths via paged attention [64], evaluating allocation granularity, occupancy limits, and end-to-end token latency.

Table 5.2 summarizes the architectural and software configuration parameters across the evaluation suite.

Table 5.2: GLMalloc evaluation configuration space across architectural axes, policies, skew models, scale, and comparative baselines.

Axis	**Evaluated Configurations**
Allocator Configurations	
Backend	**Slab** (single-level bitmap) vs. **Buddy** (multi-order tree)
Residency	**Scratchpad-Resident** (identity-mapped) vs. **DRAM-Backed** (cache)
Load Balancing	**None** (private sub-pools), **Tier 1** (intra-device rotated rerouting), **Tier 2** (inter-device region redistribution), **Tier 3** (inter-node forwarding)
Workload	
Operation Mix	**Allocate-only** vs. **Allocate-and-release**
Skew Models	Intra-device ($H \leq 32$ hot lanes), Inter-device ($U32$ hot devices), Inter-node ($H \in \{1, 2, 4\}$ hot nodes), and Gradient skew
Swizzle Policy	Interleave $C \in [2^{10}, 2^{27}]$ B; **Node-local**
Experiment Scales	
Machine Scale	1 to 32 nodes (2,048 to 65,536 lanes); weak-scaling up to 131,072 lanes
Comparative Baselines	
Centralized	Fan-in $R : A \in [1 : 1, 32 : 1]$; Lock-free sibling scan, Master-mediated (lockless), and Master-mediated (per-device lock)
Production	Host allocators (`ptmalloc3`, `TCMalloc`, `jemalloc`, `mimalloc`) and near-memory (`PIM-malloc`)

Baselines

We use **GLMalloc without its load-balancing mechanisms** as the baseline to measure each design tier's contribution to load balancing. This baseline represents the canonical private-pool design adopted by state-of-the-art multi-threaded allocators (e.g., Hoard [17],

jemalloc [34], TCMalloc [125], and mimalloc [70]) and by GPU dynamic allocators such as ScatterAlloc and Halloc [109, 3] (Section 6.2.5).

For centralized scaling evaluation, we use both an uncontended fan-in and a contended centralized manager for baselines.

5.3 Evaluation

5.3.1 GLMalloc's Scalable Two-Level Management

In this section, we evaluate GLMalloc's two-level management to prove that GLMalloc Level-One's coarse-grained segment allocation provides both flexibility and performance. We then characterize the Level-One and Level-Two throughput and latency to ground the GLMalloc's performance.

Decoupling Locality Control from Allocation

We co-design software memory allocators with physical address translation to eliminate cross-node memory traffic in distributed architectures. This subsection evaluates how exposing hardware swizzle translation parameters to runtime software enables flexible locality control.

2-Dimensional Swizzle Parameterization and Physical Placement In UDTrans, physical memory maps to a 2-dimensional (node × offset) space. Under the default block-cyclic interleaved configuration, the Level-Two shared `MemoryPool` library fixes the striping block size as a compile-time constant (e.g., 2MB pages). Therefore, $(1-1/N)$ of all sub-pool write operations (e.g., 75% on a 4-node machine and 87.5% on an 8-node machine) need to traverse the inter-node network fabric.

By exposing the swizzle translation parameter C to the Level-One allocator interface

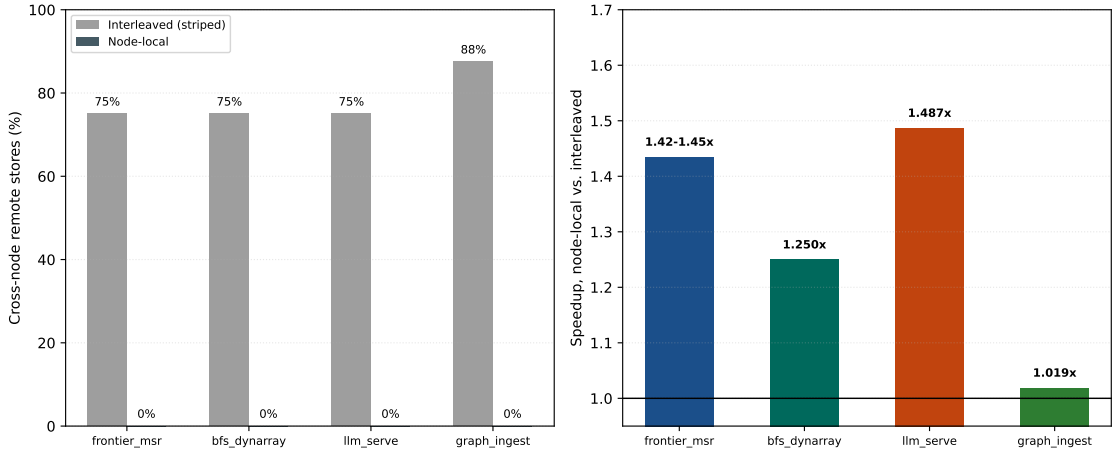


Figure 5.7: Impact of swizzle translation parameterization on cross-node memory traffic and application execution time across four distributed workloads.

(`DRAMalloc`), runtime software configures C to match each node's allocation quota:

$$C = \log_2 \left(\frac{\text{memory_pool_size}}{N} \right) \tag{5.5}$$

This assignment aligns each contiguous stripe with the allocators' pool boundary. As a result, each lane allocator's sub-pool resides within its local node DRAM, reducing remote memory stores from $(1 - 1/N)$ to 0%.

Cross-Workload Evaluation and Latency Sensitivity Figure 5.7 evaluates GLMalloc's data layout control mechanism across four application workloads from Table 5.1: `frontier_msr`, `bfs_dynarray`, `llm_serve`, and `graph_ingest` [3].

Across all four workloads, configuring node-local placement consistently achieves performance speedups ranging from 1.02× to 1.49×. While a proper config swizzle eliminates remote store to 0% in all workloads, application speedup originates from different aspects:

1. **Barrier Elimination (`frontier_msr`)**: In parallel algorithms with bulk-synchronous

3. In all configurations, Tier 3 cross-node forwarding is disabled, and workloads execute with a balanced request distribution, isolating the performance impact of physical data placement from dynamic load-balancing recovery.

parallel phases, execution time is bounded by the slowest execution lane. Under static interleaving, imbalanced remote stores across the systems create tail latency that stalls the execution at barriers. Enforcing node-local placement eliminates cross-node store delays, yielding a 1.42×-1.45× speedup across global synchronization stages.

2. **Bulk Deallocation Speedup (`bfs_dynarray`)**: In dynamic graph traversal, vertex frontier arrays are allocated incrementally and reclaimed in bulk between search levels. Under node-local placement, clearing allocated sub-pools runs entirely in local DRAM without remote invalidation, accelerating the deallocation routine by 3.39× and delivering an overall 1.24×-1.25× application speedup.

3. **Memory-Bound Serving Latency Reduction (`llm_serve`)**: In large language model serving under paged attention, Key-Value (KV) caches are continuously updated and retrieved during autoregressive token generation. Because inference throughput is bounded by memory-access latency, eliminating cross-node interconnect traversals for cache accesses delivers a 1.49× speedup (1.75× during the active token generation phase).

4. **Allocation Intensity (`graph_ingest`)**: When the memory access is high-irregular and random, localizing the memory access is hard with a single swizzle mask change, yielding a modest 1.02× **speedup**.

These findings demonstrate that exposing 2-dimensional swizzle translation parameters to software applications enables GLMalloc to enforce physical data locality as a first-class control primitive. Applications achieve near-ideal locality and avoid inter-node communication penalties without expensive page migration or program rewriting.

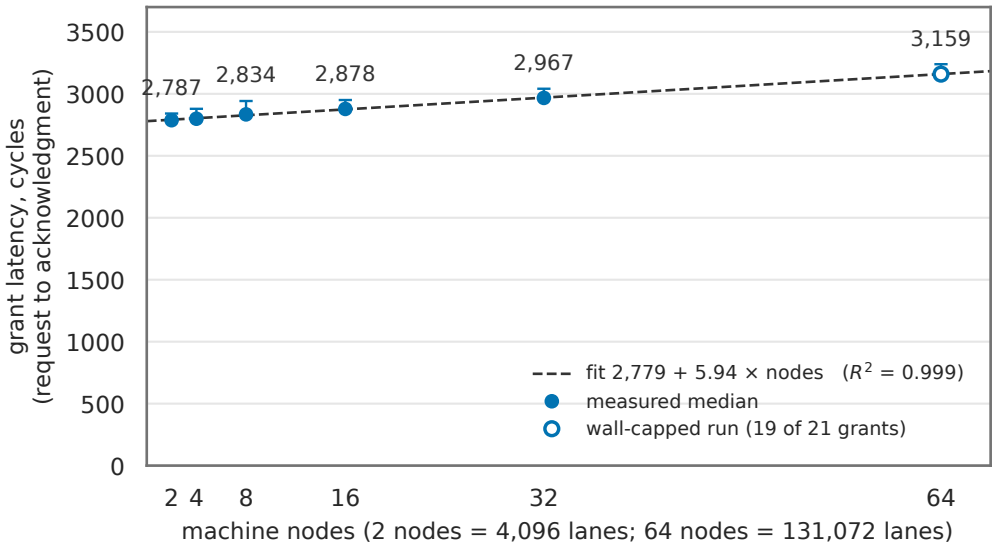


Figure 5.8: Level-One segment allocation latency across machine size. Allocation cost remains flat as the machine scale.

Level-One Allocation Performance Breakdown

Figure 5.8 plots the latency to allocate a privileged segment (via `mm_malloc_global`) across the workloads and from 2 to 64 nodes. A segment allocation costs 2,787 cycles at 2 nodes and 3,159 cycles at 64 nodes. A least-squares regression yields $T(N) = 2{,}779 + 5.94N$ cycles ($R^2 = 0.999$), indicating that each additional node adds approximately 6 cycles of inter-node broadcast and barrier acknowledgment. Tree initialization and barrier overhead account for 88% of total latency at 64 nodes; therefore, a 32× increase in node count raises allocation latency by about 13.3%. Compared to the unprivileged fast path, one privileged segment installation costs approximately five local allocations and is amortized across the application runtime and millions/billions of operations.

We count the privileged Level-One translation entry installed by each workload relative to the unprivileged Level-Two allocations they service:

- On `frontier_msr` across 4 nodes (8,192 lanes), the runtime installs **3 Level-One segments**, that is, the block pool, the partition array, and the output histogram,

serve the entire run.

- On `llm_serve` across 64 nodes (131,072 lanes), the runtime installs **2 Level-One segments**, that is, the KVCache global pool and the slab bitmap region, from which GLMalloc services 33,554,418 fine-grained page allocations.
- On `graph_ingest` across 8 nodes (16,384 lanes), the runtime installs **6 Level-One segments** (four graph arrays, the dynamic pool, and its DRAM bitmap), servicing 961,614 dynamic block allocations for neighbor adjacency lists.
- On `bfs_dynarray` at scale 22, **6 Level-One segments** serve the entire traversal, supplying 33,699 dynamic frontier-block allocations across twelve BFS tree levels.

Across all evaluated workloads, the ratio of unprivileged Level-Two dynamic allocations to privileged Level-One segments ranges from 5.6×10^3 to 3.4×10^7. Our evaluation shows that GLMalloc respects UDTrans's translation table bounds: the descriptor counts are less than 8 across system scales spanning 2,048 to 131,072 lanes, well below the UDTrans cache and translation memory size and is not scale with the machine size. Instead, segment counts scale with declared data structures.

Level-One Segment Management Throughput and Capacity Headroom Beyond the on-device broadcast latency measured in Figure 5.8, the host-side privileged manager (`DRAMalloc`) needs to resolve the underlying 2-dimensional allocation problem across physical memory capacity (m) and node location (l) [116]. `DRAMalloc` indexes free physical memory using a 2-dimensional k-ary R-tree whose bounding boxes manage rectangular node-capacity partitions across all memory nodes. Host-side evaluation over 10,000 allocation and deallocation cycles across synthetic data-intensive workloads demonstrates average decision latencies below 50 μs (Table 5.3): 19.9 μs under exponential-size graph ingestion (`Netflow`), 27.4 μs under uniform random distributions (`Random`), and 42.0 μs under adaptive mesh refinement (`AMR`). On a single host core, this corresponds to a throughput of 23,800 to 50,250

Table 5.3: Level-One segment allocation performance and two-dimensional free space fragmentation under synthetic segment-request workloads [116].

	Request Parameters				Manage	Largest	Avg. Latency
Workload	Size (blocks)	Blocks/Node	Nodes	Starting Node	Cost	Free Memory	(μs)
Random	$u(10, 1000)$	$u(1, 20)$	$u(1, 4096)$	$u(1, 4096)$	**3.6**	7.3 %	27.4
Netflow	exp(1)	$u(1, 20)$	$u(1, 4096)$	$u(1, 4096)$	47	8.5 %	**19.9**
AMR	$\times 2/10$	$u(1, 20)$	$u(1, 4096)$	$u(1, 4096)$	52	8.5 %	**42.0**

segment allocations per second ($\geq$ 20,000 req/s), scaling to millions of operations per second across a multi-threaded server. This throughput exceeds the capacity of distributed cluster coordinators (e.g., GoogleFS and ZooKeeper, which service 500-10,000 authenticated control operations per second).

In this section, we show that because each Level-One segment corresponds to a major application data structure, the cost can be amortized over application runtime and the fine-grained Level-Two allocations, a single-threaded Level-One throughput of $>$ 20,000 allocations/s provides more than four orders of magnitude of headroom above the application requirements. Furthermore, R-tree management overhead remains bounded between 3.6 and 52 tree nodes per live allocation, and free-space fragmentation stabilizes at 7.3%–8.5%, demonstrating that privileged Level-One management introduces neither a throughput bottleneck nor physical fragmentation.

Level-Two Allocation Performance Breakdown

Figure 5.9 presents the latency breakdown of GLMalloc's fast path allocation across 2,048 lanes on a node. When the requester and allocator are on the same lane, allocations are served in less than 279 cycles across allocator backends and residency schemes (Fixed-Size Slab, Scratchpad-Resident Buddy, and DRAM-Backed Buddy). Because each execution lane exclusively owns its sub-pool allocation bitmap, fast-path allocation eliminates the expensive spinlocks, compare-and-swap (`CAS`) operations, or inter-lane synchronization traffic. This

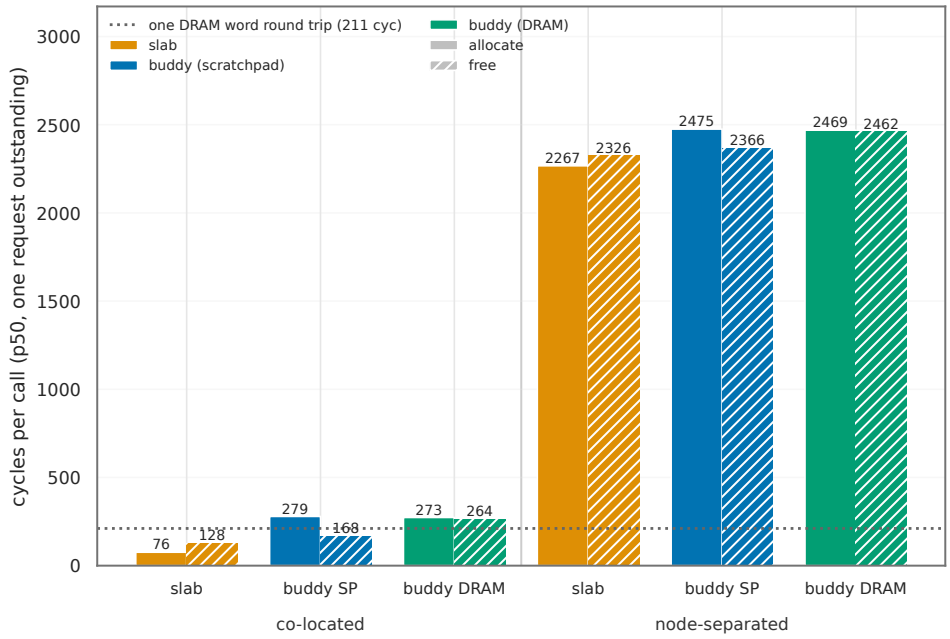


Figure 5.9: Allocation and deallocation latency of a single outstanding request across both allocator backends and residency policies and physical placements, compared against a local DRAM word round trip (400 cycles dash line).

latency corresponds to pure local execution and does not involve any DRAM traffic.

Figure 5.11 illustrates the latency changes when the sub-pool fullness varies from 0% to 120% on the Fixed-Size Slab backend. Local bitmap search is occupancy-invariant: allocation latency remains 85 cycles as occupancy increases from 0% to 99%, except for the first request, which populates the bucket cache. Beyond 100%, overflow requests transition to intra-device rerouting at 160 cycles to fulfill every allocation request. In contrast, disabling rerouting causes all overflow requests to fail.

Metadata and Memory Footprint Table 5.4 characterizes metadata overhead. On the Fixed-Size Slab backend, metadata is 1 bit per block in the tracking bitmap plus 800 bytes of scratchpad control state per local allocator. As for the buddy backend in scratchpad-resident mode, tracking multi-order power-of-two trees requires 7 bits per block in scratchpad SRAM, remaining within 0.34% of capacity for 256 B base blocks.

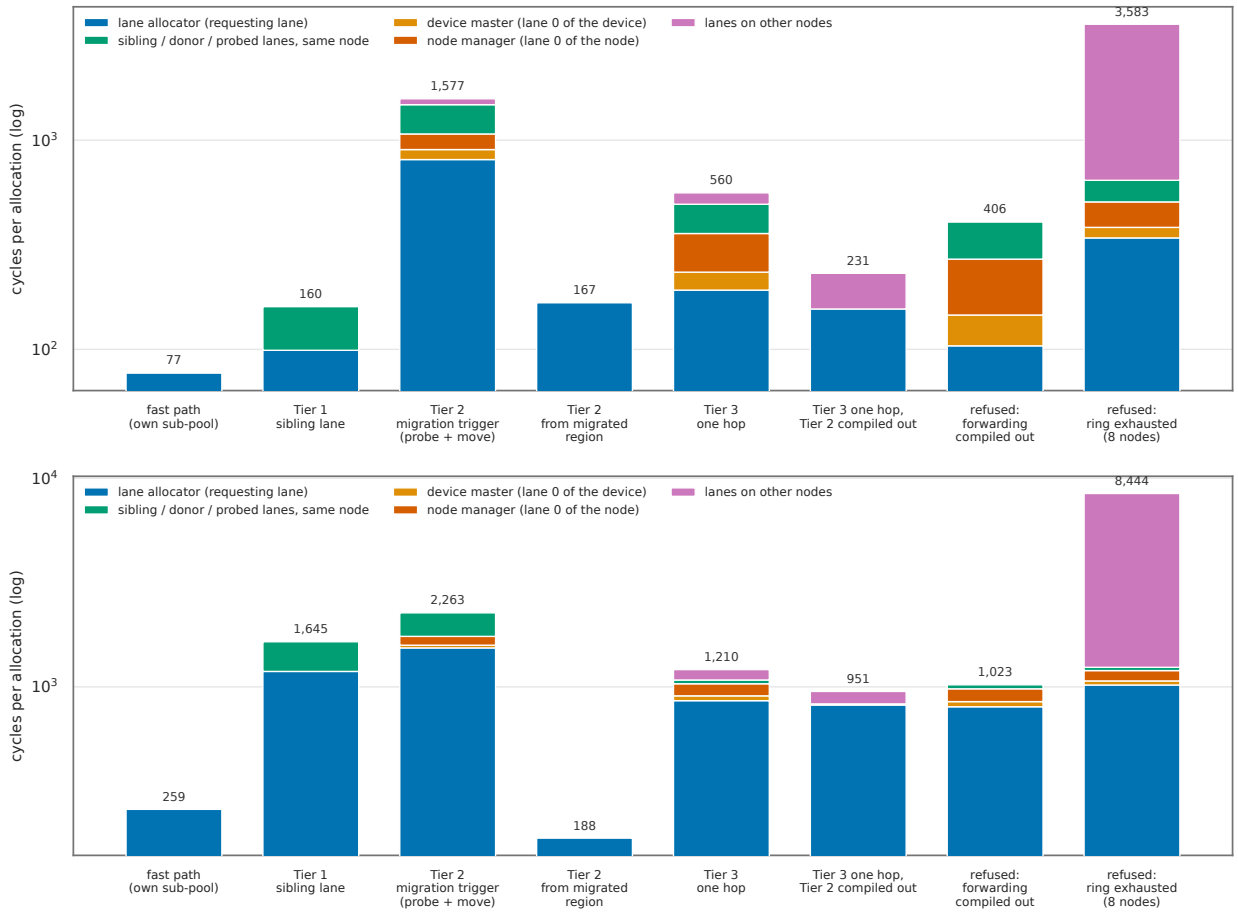


Figure 5.10: Cost of one allocation at each tier of the allocation path (top: slab backend, bottom: buddy backend).

5.3.2 GLMalloc Level-Two's Decentralized, Skew-Resilient Hierarchical Load-Balancing

While static per-lane sub-pooling provides lock-free fast-path execution, it is vulnerable to spatial/temporal demand skew: lanes processing dense data subsets may exhaust their private sub-pools while neighboring lanes still have abundant capacity, or a sudden burst of allocation to a lane may overflow its incoming request queue. This subsection evaluates GLMalloc Level-Two's unprivileged 3-tier load balancing across system scales under synthetic, adversarial, and data-driven skew distributions.

We first characterize the performance at each tier and evaluate the cost when requests

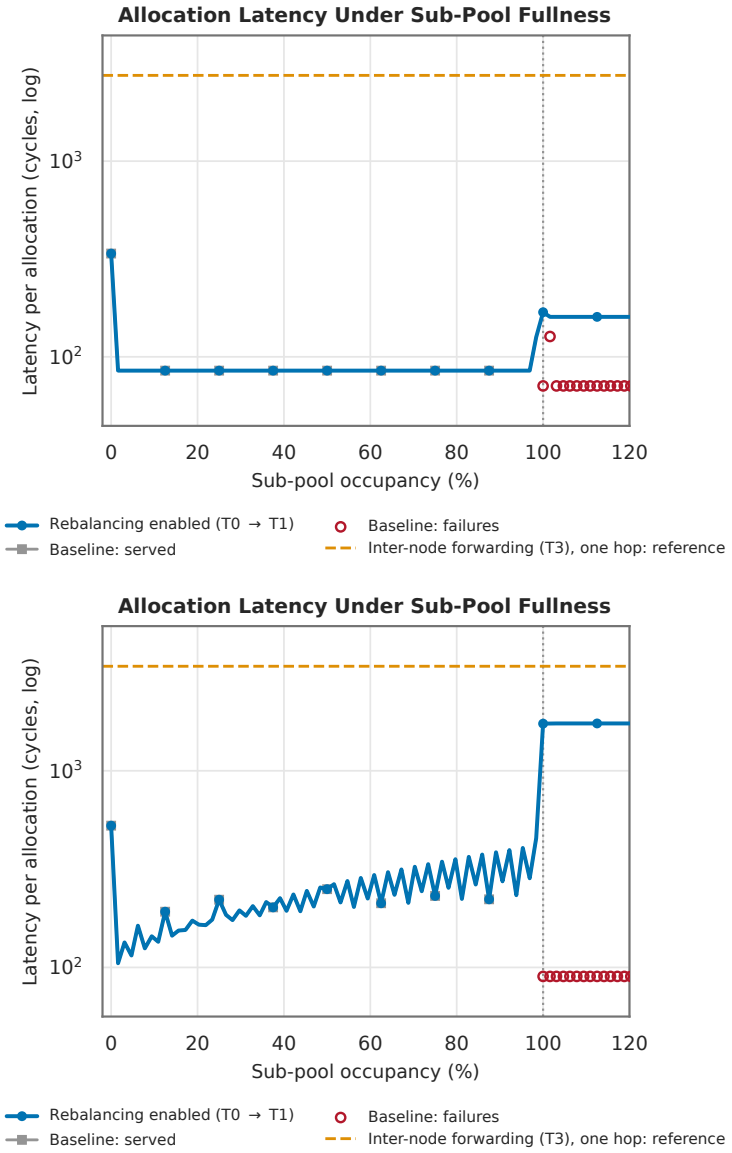


Figure 5.11: Allocation latency under various sub-pool occupancies(top: slab backend, bottom: buddy backend). When load balancing is disabled, overflow requests fail unconditionally (red open bullets).

Table 5.4: GLMalloc Level-Two's metadata size across backends and configurations.

Allocator & Data Structure	Metadata State	Explanations
Slab DRAM bitmap	1 bit/block	1 word per 64 blocks in DRAM
Slab Scratchpad per-lane	800 B/lane	Amortizes to 0.2–12.5 B / block
Slab Device managers	192 B/device	Each device's lane 0 scratchpad
Buddy Per-order bitmaps in DRAM	7 bits/block	E.g., orders 0-6 for 64 blocks per lane
Buddy Scratchpad master state	640 B/device	Each device's lane 0 scratchpad

cross physical node boundaries, demonstrating that cross-node request forwarding fully recovers capacity but incurs expensive remote memory-traversal. This cost motivates robust intra-node load balancing. We then evaluate Tier 1 intra-device lock-free rerouting, compare it against centralized shared-state alternatives, and characterize the Tier 2 inter-device region redistribution within the node.

3 Tier Load Balancing Basic Performance Characterization

Figures 5.10 reports the cost (execution cycles) of allocation at each tier of GLMalloc's allocation path for the fixed-size slab and variable-sized buddy backend, respectively. The bars are divided by the work performed: the allocator lane, the device master, the node manager, other lanes on the same node (the serving sibling, the donor, and probed lanes), and lanes on other nodes.

- **Fast Path (Own Sub-Pool)**: The request is served from the scratchpad-cached bitmap, in 259 cycles on the buddy and 77 cycles on the slab.
- **Tier 1 (Sibling rerouting)**: The request costs 1,645 cycles on the buddy, of which 1,186 execute on the requesting lane and 459 on the sibling that serves the block, and 160 cycles on the slab. The buddy's cost is dominated by donor selection : the cost on the requesting lane is 178 cycles plus 15.7 cycles per lane examined.

- **Tier 2 (Migration Trigger)**: The request that triggers a region migration costs 2,263 cycles on the buddy and 1,577 cycles on the slab, paid once per migration.
- **Tier 2 (Migrated Region)**: Subsequent requests are served from the migrated region by the requesting lane alone, in 188 cycles on the buddy and 167 cycles on the slab.
- **Tier 3 (One Hop)**: Forwarding the request to a neighbouring node costs 1,210 cycles on the buddy and 552 cycles on the slab.
- **Refused Request**: With forwarding compiled out, a request that exhausts its node is refused after 1,023 cycles on the buddy and 398 cycles on the slab. When every node is full, the request visits all nodes before it is refused, at 8,444 and 3,519 cycles, because each node visited repeats the allocator, node manager, device master, and probe work of the requesting node.

The slab backend executes fewer cycles than the buddy at every tier, by a factor of 3.4 on the fast path, 10.3 on Tier 1, 2.2 on Tier 3, and 1.43 on the two Tier 2 cases.

The Inter-Node Imbalance Boundary and the Motivation for Intra-Node Balancing

Section 5.3.1 showed that physical memory locality is crucial to performance in distributed architectures: remote DRAM accesses incur higher latency and lower bandwidth. When skewed demand hits a physical node and all 32 local devices exhaust their private sub-pools, load balancing needs to cross node boundaries, which is what Tier 3 load balancing is supposed to do.

Uncover the Node Sub-Pool Capacity Limit Figure 5.12 characterizes this boundary on the `node_skew` synthetic suite across an 8-node machine (16,384 lanes) under varied hot nodes $H \in \{1, 2, 4\}$ and hot/cold demand quotas. Without Tier 3, delivered capacity on the

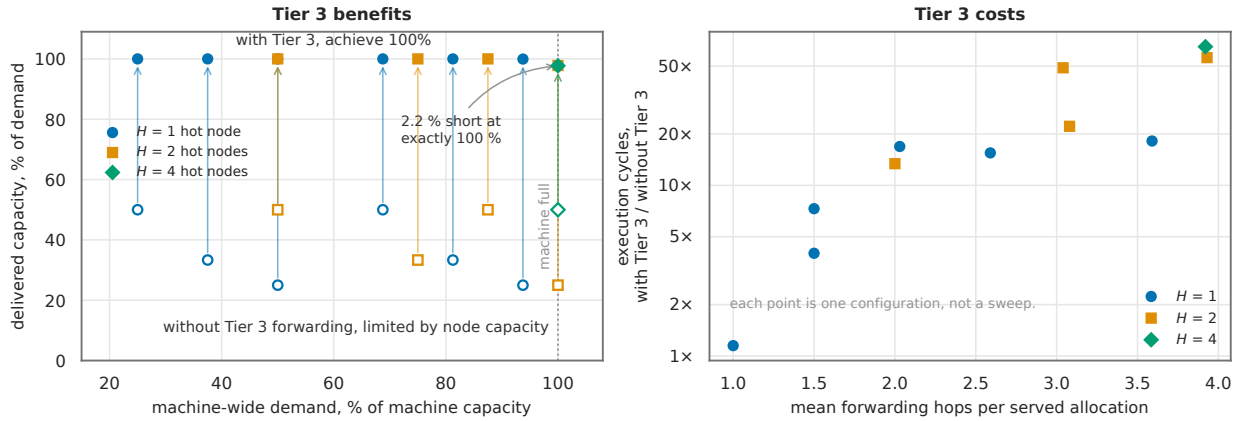


Figure 5.12: Tier 3 inter-node request forwarding. **Left**: delivered requests with respect to demand. With Tier 3 (filled), the requested memory that can be served equals the global capacity. **Right**: the price of Tier 3: Each additional forwarding hop costs cycles, from 1.15× with 1 hop to 65× with 3.9 hops.

hot node is capped at the theoretical ceiling $\min(1, 100/\text{hot})$ in every cell (e.g., delivering 50% requests at 200% demand, 33% at 300%, and 25% at 400%).

Enabling Tier 3 request forwarding removes this ceiling: all allocation requests are served where the demand is within global machine capacity ($H \cdot \text{hot} + (8 - H) \cdot \text{cold} < 800\%$). The modest $\approx 2.2\%$ shortfall observed in three boundary cells ($H = 1$ 400/50, $H = 2$ 400/0, and $H = 4$ 200/0) occurs at 100% machine-wide utilization, where donor nodes' Tier 2 admission ceilings turn away some forwarded requests while other donor nodes still retain spare capacity.

Figure 5.12 illustrates the trade-off that guides the design of GLMalloc Level-Two management: **Tier 3 recovers capacity at a cost of locality**. Each forwarded allocation is fulfilled in another node's physical DRAM, so subsequent application accesses need to go through the inter-node network. Therefore, total runtime grows by 1.15× to 65.0× relative to the baseline as the remote memory accesses and inter-node communications rise.

Segment Layout vs. Rerouting Policy One hypothesis we evaluated is whether modifying the swizzle interleave granularity C of the segment can alleviate cross-node memory

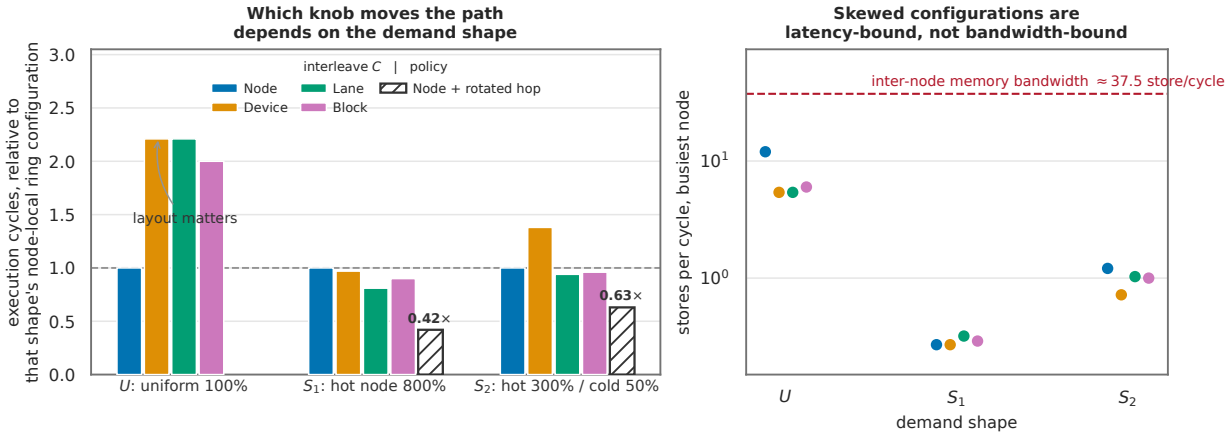


Figure 5.13: Segment layout and Tier 3 rerouting. **Left**: execution cycles for every layout, normalized to node-local. **Right**: skewed applications are latency bound.

contention under remote service. Figure 5.13 examines this hypothesis across swizzle interleaves from node-local ($C = 2^{27}$) down to block-striped ($C = 2^{10}$) under uniform and skewed demand shapes.

Under uniform demand (U), the node-local layout reproduces the findings of Section 5.3.1, it outperforms striped configurations by 2.00×–2.21× by eliminating remote DRAM stores. Under hot-node skew (S_1 and S_2), however, altering C buys back negligible performance (0.81-0.97× relative to node-local). This occurs because the allocation and access path is latency-bound: the busiest node processes only 0.27-1.21 DRAM stores/cycle, far below the ≈ 37.5 store/cycle DRAM bandwidth cap.

Instead, the dominant performance bottleneck is donor-node allocator contention. Sequential ring routing sends all 2,048 overflowing lanes to the identical neighbor node simultaneously, overwhelming that node's Tier 1/2 admission queues and triggering retry stalls. In contrast, rotating the first forwarding hop (`rot`) per lane distributes donor requests across all seven neighbor nodes while maintaining node-local layout. As shown in Figure 5.13, hop rotation reduces execution time to 0.42× on S_1 (12.0 M vs. 28.8 M cycles) and 0.63× on S_2 (4.0 M vs. 6.3 M cycles) while resolving all observed allocation failures.

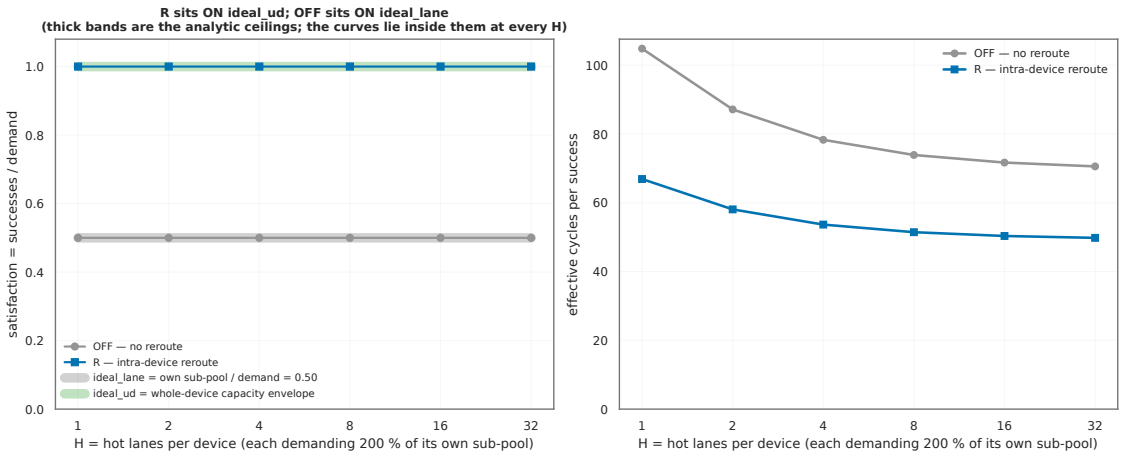


Figure 5.14: Delivered capacity and execution cost under intra-device load skew. Rotated rerouting achieves 100% capacity utilization and reduces cost per successful allocation by 29–36%.

The Importance of Robust Intra-Node Load Balancing These findings above shape the design of the GLMalloc Level-Two management: because crossing the physical node boundary carries a substantial latency penalty, the memory manager should absorb skewed requests locally within the node for performance. Cross-node forwarding guarantees against job abortion under whole-node oversubscription, but routine load balancing should operate intra-node. We now evaluate GLMalloc's two intra-node mechanisms: Tier 1 intra-device rotated rerouting and Tier 2 inter-device region redistribution.

Intra-Node Hierarchical Balancing: Tier 1 Rerouting and Tier 2 Redistribution

Tier 1 Rotated Rerouting To isolate the algorithmic contribution of the load-balancing mechanisms, we use GLMalloc without load balancing as the baseline, representing the canonical private-pool discipline adopted by allocators with static heap assignments (e.g., glibc's fixed pool/arena mapping [43] and PIM-malloc [69]).

As shown in Figure 5.14, intra-device rerouting not only doubles delivered capacity but also reduces the execution cost per successful allocation by 29%–36% compared to the unbalanced baseline. Rerouting eliminates the expensive failure-signaling and retry loops in

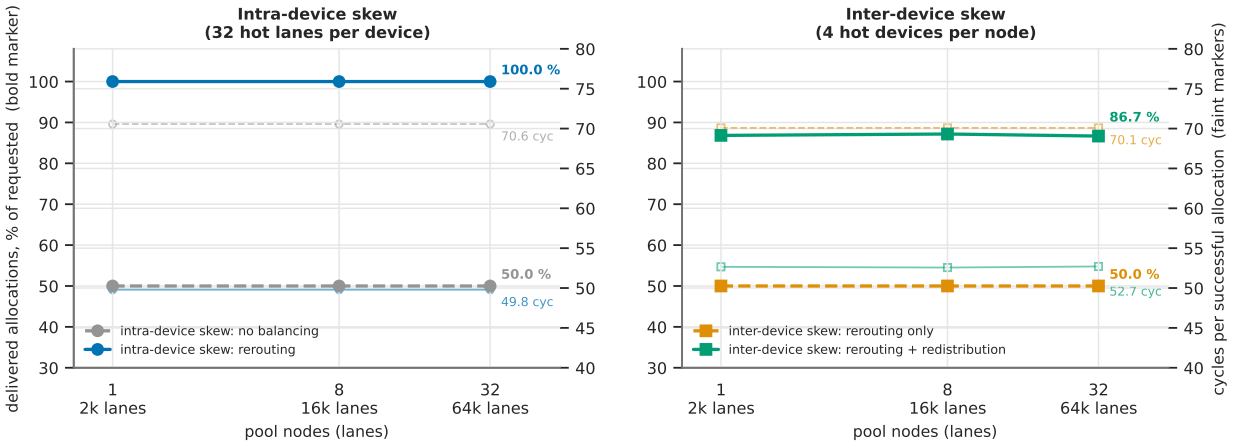


Figure 5.15: Load-balancing capacity and execution cost under machine scaling from 1 to 32 nodes (2,048 to 65,536 lanes) with per-device demand and per-node demand skew.

the baseline. In return, rerouting increases instruction count (338.6 instructions per success compared to 111.8 for the unbalanced baseline.

Figure 5.15 scales the machine from 1 node (2,048 lanes) to 32 nodes (65,536 lanes) while holding the skew fixed. Under intra-device skew (Figure 5.15 left), the unbalanced baseline delivers only 50.0% of capacity at 70.6 cycles per success, while Tier 1 rerouting delivers 100% of capacity at a cost of 49.8 cycles per success. The cost remains the same from 2,048 lanes to 65,536 lanes because Tier 1 only requires data shared within a device via a local availability vector and never leaves the device, eliminating costly cross-device synchronization that grows with machine size. Under inter-device skew (Figure 5.15 right), the Tier 1 only can only deliver 50% of the demand, whereas enabling the Tier 2 raise it to 86% and reduces the cost per successful allocation from 70.1 cycles to 52.7 cycles.

GLMalloc Level-Two Rerouting vs. Centralized Shared-State Mediation and Mutual Exclusion

In this section, we evaluate the scalability of GLMalloc Level-Two memory management with decentralized, peer-to-peer rerouting and compare it to conventional centralized shared-state

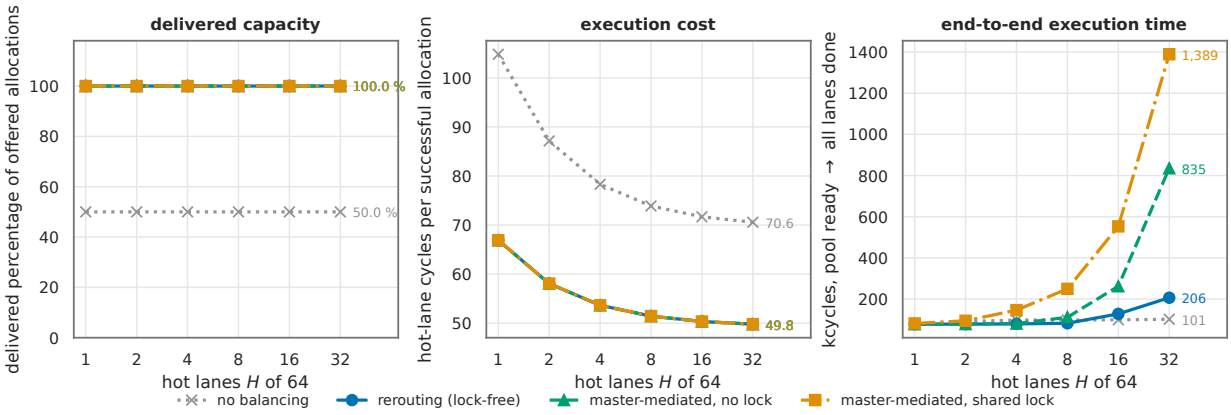


Figure 5.16: The comparison between centralized versus decentralized rerouting under intra-device skew ($H \in [1, 32]$ hot lanes of 64 per device, each demanding 200% of local pool capacity).

manager. To isolate the root causes of concurrency degradation, Figure 5.16 evaluates four controlled architectural configurations executing identical allocation workloads under intra-device skew ($H \in [1, 32]$ hot lanes demanding 200% of local pool capacity):

1. **Isolated Private Pool (No Balancing)**: lanes allocate solely from local sub-pools without cross-lane capacity mobility.

2. **Decentralized Rotated Rerouting (GLMalloc Tier 1)**: Exhausted lanes directly inspect peer scratchpad state via unprivileged, lock-free rotational traversals.

3. **Master-Mediated Coordination (Uncontended Coordinator)**: Reallocations are queued to a dedicated coordinator lane, which reassigns capacity without mutual exclusion.

4. **Master-Mediated Mutual Exclusion (Shared Lock)**: Allocation is serialized through a central coordinator under a global device mutex lock.

Capacity and Execution Time As shown in Figure 5.16 right, all three load-balancing-enabled configurations achieve 100% delivered capacity across varied skewness. However, while capacity recovery is uniform, total device completion time diverges substantially. At

$H = 32$, allocating requests through a centralized coordinator expands device completion time from 206.0 k cycles to 835.1 kcyc (a 4.05× slowdown), even in the absence of lock contention. Imposing mutual exclusion via a shared device mutex further compounds device completion time to 1,389.1 kcyc (6.74× slowdown).

Decomposing this latency penalty uncovers the sequential escalation cost of centralized management:

$$\text{Slowdown}_{\text{total}} = \underbrace{4.05\times}_{\text{Centralized Mediation}} \times \underbrace{1.66\times}_{\text{Mutual Exclusion}} = 6.74\times \tag{5.6}$$

Centralized coordination alone accounts for 71% of the total latency expansion. The coordinator operates as a single-server queue with finite service rate μ; as concurrent request escalation emerges beyond $H \geq 4$, the arrival rate exceeds service throughput ($\lambda > \mu$), causing queuing delays to scale linearly with H. At $H \leq 4$, mediation overhead remains below 2%, but performance degrades steeply as hot-lane concurrency increases. Therefore, eliminating software locks without eliminating centralized coordination fails to prevent concurrency collapse.

Tier 2 Inter-Device Redistribution and Saturated Device Pool Recovery Under inter-device skew (Figure 5.17), intra-device rerouting cannot cross device boundaries and is bounded by the capacity per lane. Enabling Tier 2 inter-device region redistribution raises delivered capacity to 96.1%–99.4% on slab and 88.3%–94.7% on buddy.

Under inter-device skew (Figure 5.17), intra-device rerouting cannot cross device boundaries, so it is bounded by the capacity of the hot device itself: it delivers 50.0% of the requested allocations at every U. Enabling Tier 2 inter-device region redistribution moves free regions from cold devices into the hot ones and raises delivered capacity to 73%–89% on the slab backend and 64%–88% on the buddy backend. Conversion declines as U grows, because more hot devices compete for the node's bounded migration concurrency and swap

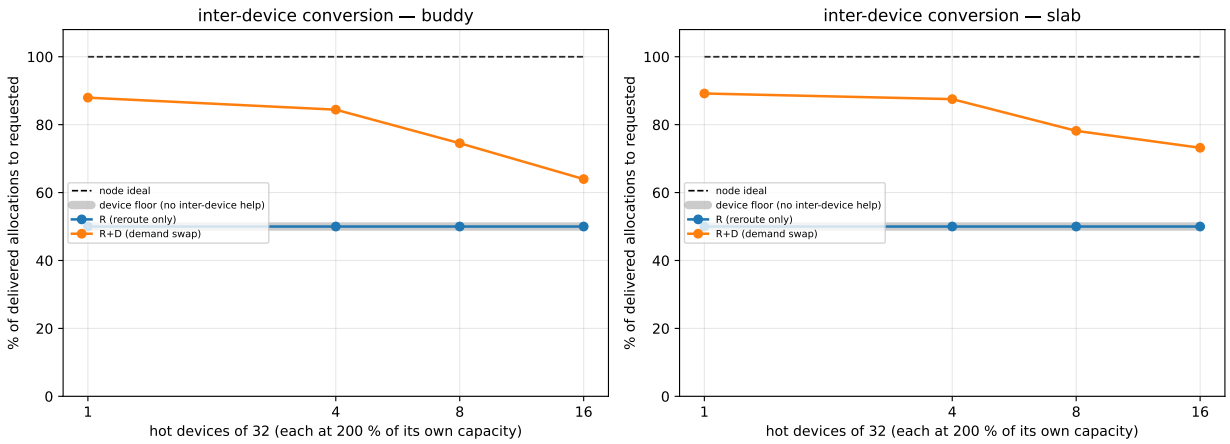


Figure 5.17: Capacity conversion under inter-device load skew (U of the node each receive 200% of their own capacity).

budget, and the decline is steeper on the buddy backend than on the slab backend.

Load Balancing Under Skew

In this section, we evaluate GLMalloc 3-tier load balancing performance in real-world workloads,

Figure 5.18 evaluates inter-device request migration on the `frontier_msr` workload with 1 saturated device among 32 in a node. Without migration, delivered capacity collapses due to the oversubscription, while latency surges to 5,523.1 cycles. Enabling demand-driven migration recovers 100.0% delivered capacity across all oversubscription levels, bounding the allocation latency to 1,508-1,531 cycles by delivering requests from successful region migrations.

Figure 5.19 evaluates the Tier 1 and Tier 2 load-balancing mechanisms on the `bfs_dynarray` workload ($\theta = 0.8$, one node) when the pool is halved from 128 MB to 64 MB. With the 128 MB pool every allocation succeeds across the 3 configurations. With the 64 MB pool, no balancing serves only 23.3% of allocation. Intra-device rerouting lowers the success rate to 20.1% and increases the measured phase by 1.89×: once every lane of the device is full, each

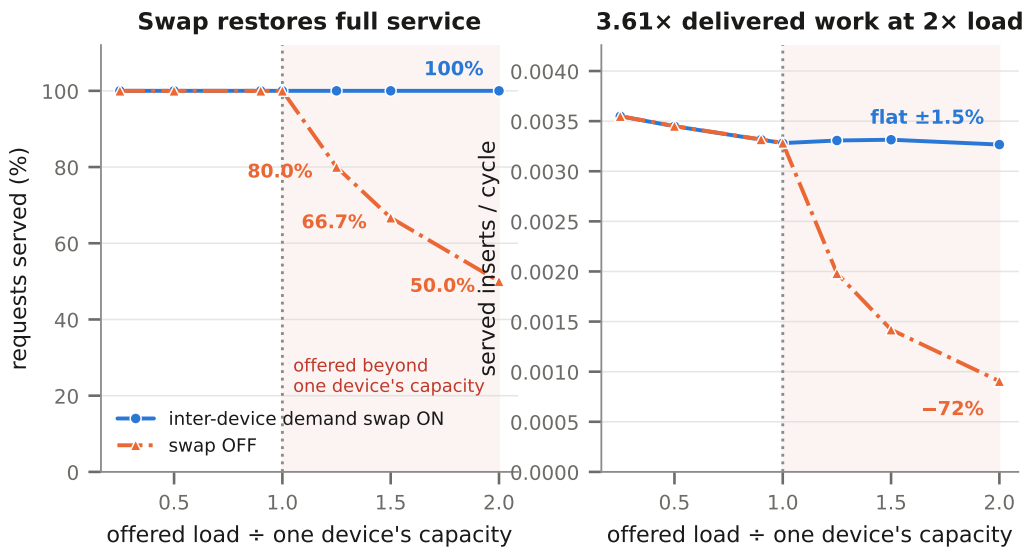


Figure 5.18: Inter-device request migration recovering full delivered capacity on the `frontier_msr` pipeline under device oversubscription.

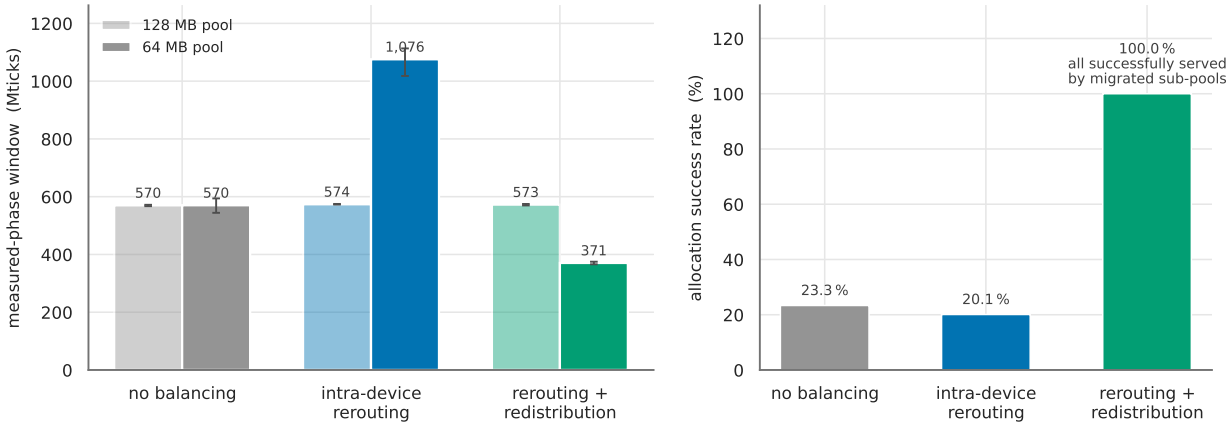


Figure 5.19: Execution time and allocation success rate on `bfs_dynarray` across 1, 2, and 8 nodes.

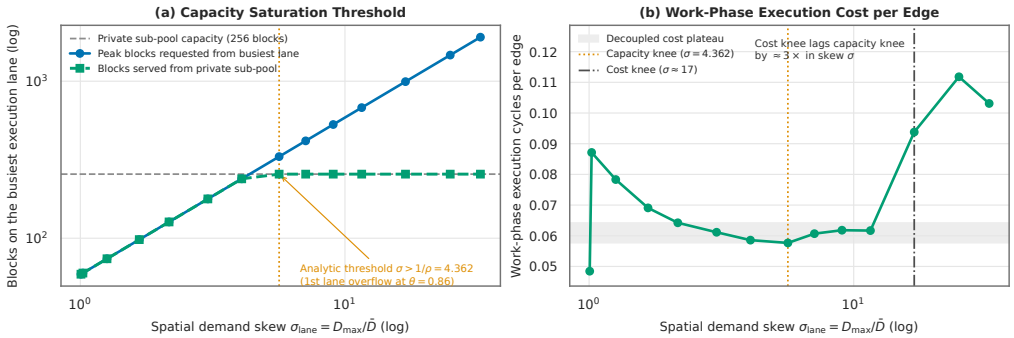


Figure 5.20: The skew threshold on `graph_ingest` (16,384 lanes). The busiest lane crosses private capacity at $\theta = 0.86$, where $\sigma_{\text{lane}} > 1/\rho = 4.362$. Rerouting absorbs all overflow, and execution cost remains flat as capacity grows.

failing request also probes the sibling lanes before it is refused, which raises the number of failed attempts from 54,555 to 66,119. Adding Tier 2 inter-device redistribution serves every request through migrated sub-pools and completes in 371 M ticks. This is 1.54× faster than no balancing and 2.90× faster than intra-device rerouting alone.

Finally, we evolute the `graph_ingest` workload across 8 nodes (16,384 lanes) using power-law degree distributions. Vertices require $\lceil \deg(v)/\text{capacity} \rceil$ blocks, and edge processing maps to assigned lanes. Sweeping the Zipf exponent $\theta \in [0.65, 0.99]$ sweeps spatial demand skew σ_{lane} from 3.0 to 33 across an aggregate footprint of 961,614 blocks.

Across all fourteen evaluated configurations, intra-device rerouting delivers every allocation request successfully, with execution cost remaining within 0.058-0.062 cycles per edge from $\theta = 0.80$ to 0.90.

Given initial pool occupancy $\rho = 0.2293$, an individual lane's demand exceeds its private sub-pool capacity when:

$$\sigma_{\text{lane}} > \frac{1}{\rho} = \frac{1}{0.2293} \approx 4.362 \tag{5.7}$$

Figure 5.20 shows that at $\theta = 0.83$ ($\sigma_{\text{lane}} = 4.072$), zero lanes exceed capacity and zero blocks reroute. At $\theta = 0.86$ ($\sigma_{\text{lane}} = 5.640$), one lane crosses the threshold, triggering the rerouting. Past this threshold, local allocations saturate at 256 blocks (the sub-pool limit), while peak

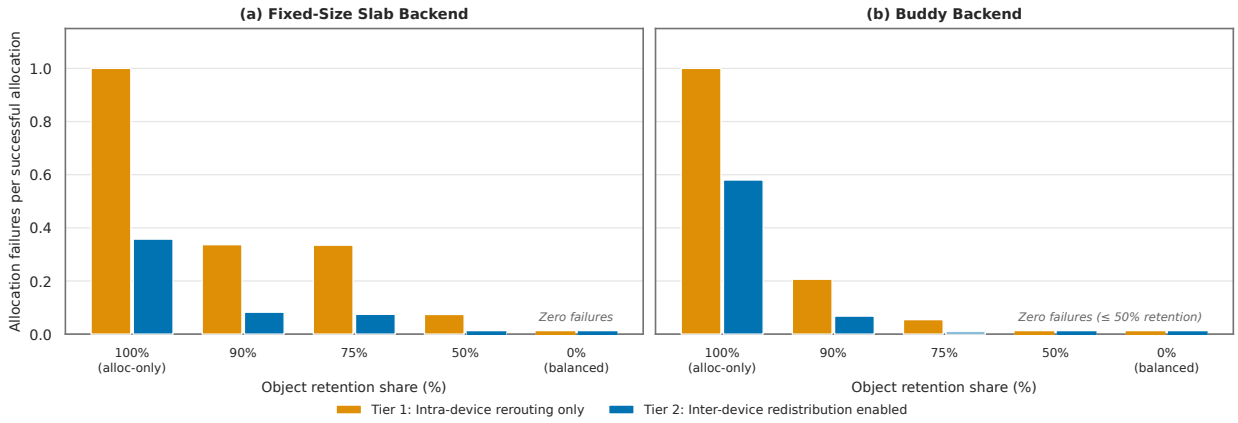


Figure 5.21: Failure rate under dynamic allocate-and-free as a function of retention.

demanded blocks climb to 1,908. Allocation cost remains decoupled from capacity binding: per-edge cycles remain flat through the capacity knee, rising only at $\sigma \approx 17$ (0.094 cycles) and $\sigma \approx 33$ (0.103 cycles).

Allocate-and-release Under Dynamic Request Streams

Figure 5.21 evaluates allocate-and-release performance by sweeping retention under skewed allocation streams. When every allocated block is retained (alloc-only), rerouting alone fails once for every successful allocation on both backends; demand redistribution lowers this to 0.36 on slab (2.8× reduction) and 0.58 on buddy (1.7× reduction). At 90% retention, demand redistribution reduces failures per successful allocation from 0.34 to 0.08 on slab (4.1× reduction) and from 0.21 to 0.07 on buddy (3.0× reduction). At 75% retention, failures drop from 0.33 to 0.07 on slab (4.5×) and from 0.054 to 0.008 on buddy (6.5×). At 50% retention, rerouting eliminates almost all failures on slab and buddy. Under pure allocate-and-release, deallocations refill allocated blocks in the owning sub-pools, so allocation requests can be served directly from freed blocks, eliminating allocation failures on both backends.

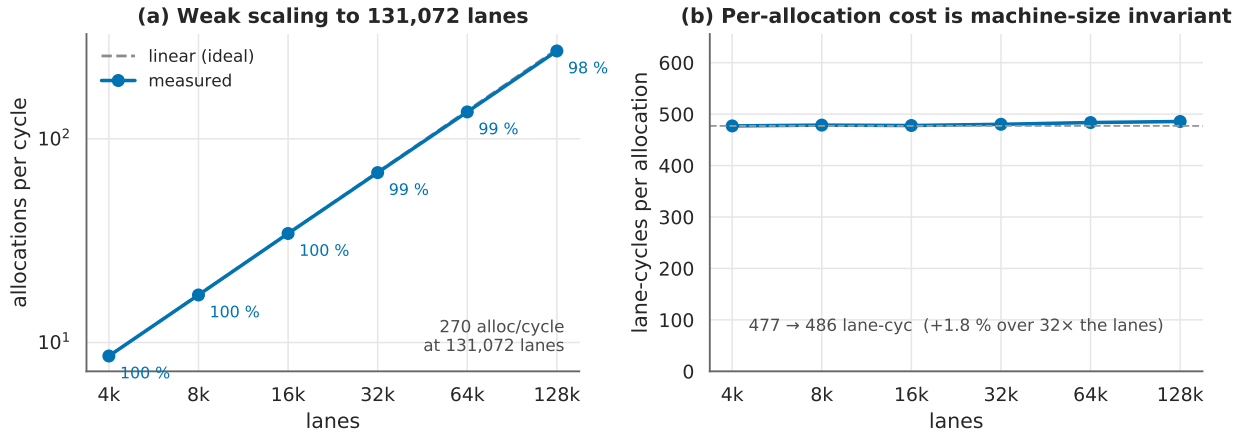


Figure 5.22: Weak-scaling throughput and per-lane execution cost for `llm_serve` from 2 to 64 nodes (4,096 to 131,072 lanes).

5.3.3 Real-World Application Performance: Scalability, Flexibility and Efficiency

This subsection evaluates GLMalloc's weak and strong scaling up to 131,072 lanes demonstrates end-to-end efficiency, and evaluates the benefits of letting applications choose between the two allocator backends across real-world workloads.

Allocator Scalability

In Figure 5.22, we measure the weak scaling of `llm_serve`. As machine and concurrency scale by 32×, throughput increases from 8.59 to 269.88 allocations per processor cycle, i.e., a 31.43× speedup, or 98% weak-scaling efficiency. Meanwhile, the execution cost per page allocation remains flat, increasing by only +1.8% while successfully allocating and releasing all 33.55 million pages. Strong scaling on the same server across an 8× node expansion reaches 83% efficiency (6.61×).

Figure 5.23 measures the strong scaling of `llm_serve` with the total work held at 8.39 million page allocations. The parallel efficiency remains high up to 16 nodes: throughput increases from 10.33 to 68.25 allocations per processor cycle, a 6.61× speedup over an 8×

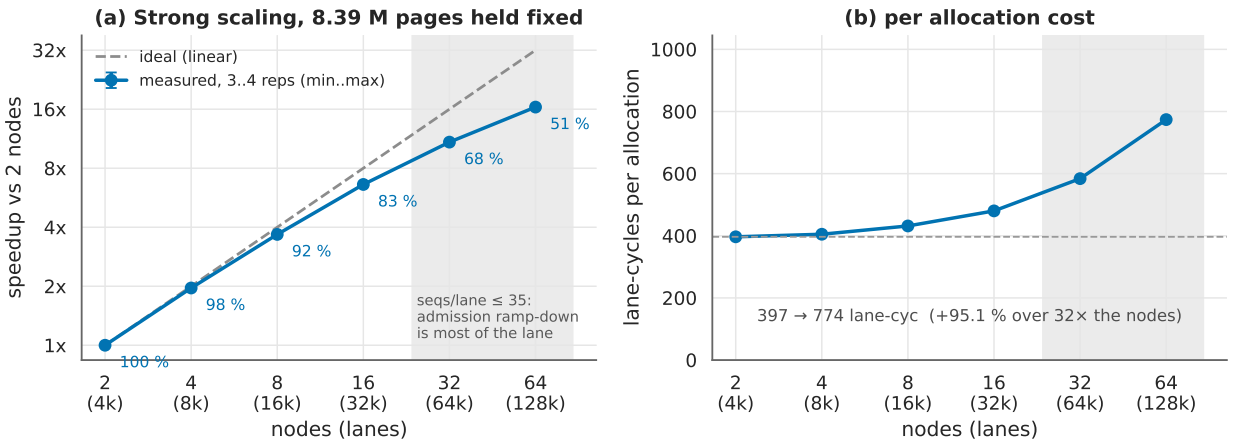


Figure 5.23: Strong-scaling throughput and per-lane execution cost for `llm_serve` from 2 to 64 nodes (4,096 to 131,072 lanes).

node expansion. Beyond 16 nodes, the efficiency declines to 68% at 32 nodes and 51% at 64 nodes (16.40×, 169.36 allocations per cycle), because each lane then serves only 32 and 16 sequences. Similarly, the execution cost per page allocation rises from 397 to 480 lane-cycles (+21%) up to 16 nodes and to 774 lane-cycles (+95%) at 64 nodes. The loss therefore reflects the diminishing work per lane rather than a scaling limit of the pool.

Centralized Manager Figure 5.24 contrasts GLMalloc's decentralized allocation against centralized allocator schemes. As requester-to-allocator fan-in concentrates from 1 : 1 (decentralized sub-pooling) to 32 : 1 (a single device manager), aggregate device throughput collapses from 5,244.1 to 123.6 operations per 1,000 cycles; that is, a 42.4× performance reduction due to a 39× growth in the request queue. The throughput per allocator lane drops by up to 1.50×.

Centralized Coordination While the fan-in experiment above isolates the throughput loss from the centralized request queue, each allocator lane still manages a private sub-pool bitmap without inter-lane synchronization. In a fully centralized memory allocator, multiple memory requesters not only share the allocator threads but also need to synchronize on a

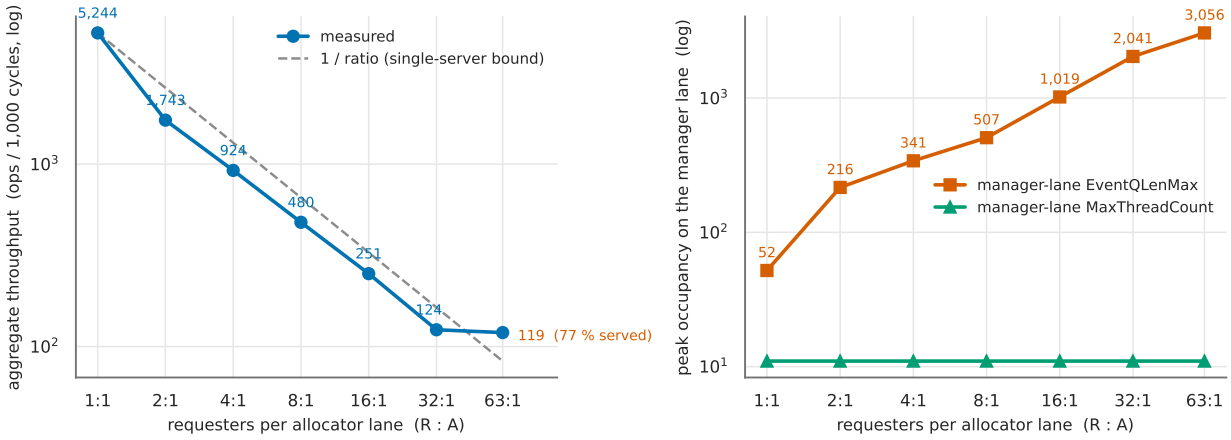


Figure 5.24: The cost of concentrating requests. Concentrating 32 requesters onto a single allocator lane reduces aggregate device throughput by 42.4× and surges request queues by 39×.

single shared free-memory structure (e.g., a centralized free list, global heap, or allocation lock), adding coordination overhead.

To characterize this state-sharing bottleneck, Figure 5.25 evaluates a centralized allocator where all allocation operations serialize on a single allocation structure per node. A spin-requeue lock keeps waiting requests in the queue as the lock is released, so dequeue latency scales with waiter queue depth, substantially constraining throughput. To upper-bound the performance, our implementation uses a direct hardware wait queue, ensuring constant-time lock transfers without queuing overhead.

The results show that sharing allocation state fails to scale because of limited resources and queuing contention. Across all allocator lane configurations ($A = 1$ to 32 allocator lanes serving $R = 32$ requesters per device), total system throughput remains 1 operation per 1,000 cycles. Provisioning additional allocator lanes doesn't yield throughput improvement because a single shared lock serializes every request.

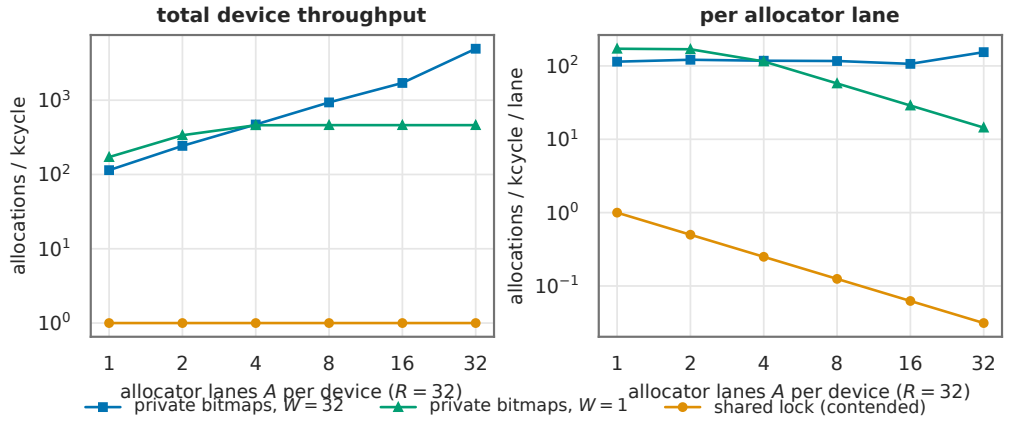


Figure 5.25: Aggregate throughput under a single shared allocation structure (contended manager) versus private per-lane sub-pooling across allocator lane counts $A \in [1, 32]$.

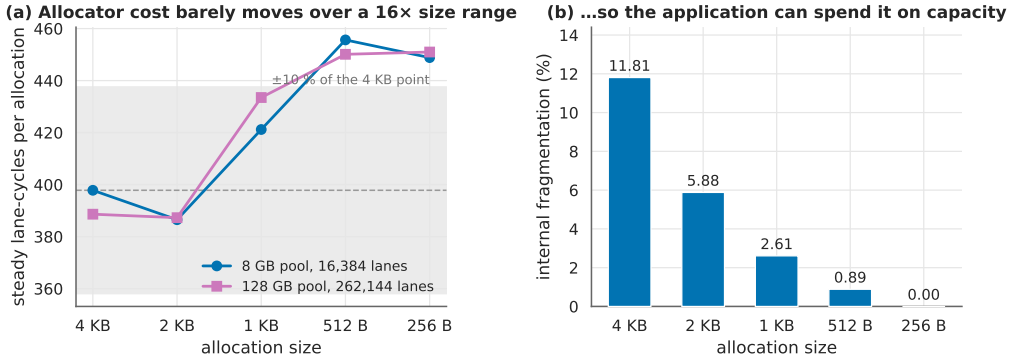


Figure 5.26: Allocation latency across a 16× block size granularity sweep on `llm_serve`, varying by only 12.8% between 256 B and 4 KB.

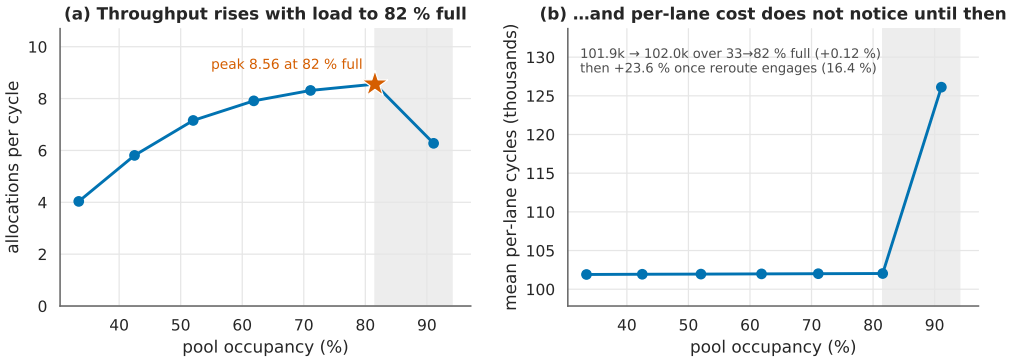


Figure 5.27: System throughput as a function of pool occupancy on `llm_serve`, identifying the operational admission knee at 82% occupancy.

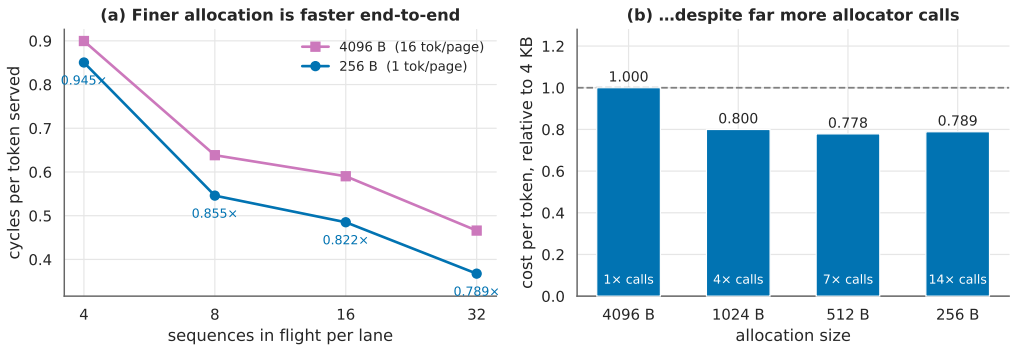


Figure 5.28: End-to-end token generation latency on `llm_serve` as a function of block size. Sizing blocks to 512 B achieves 22% faster token processing than 4 KB blocks by eliminating zero-padded KV cache memory traffic.

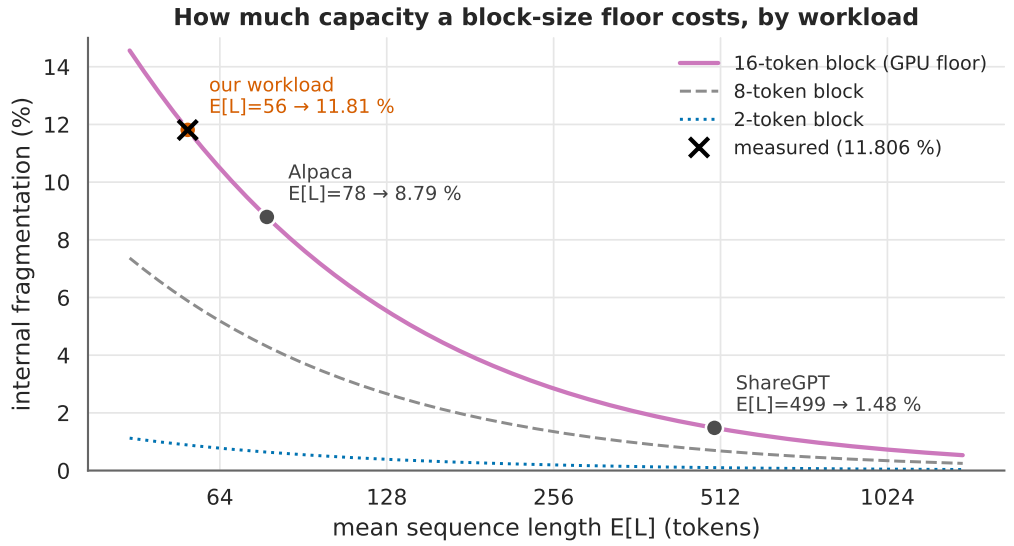


Figure 5.29: Internal memory fragmentation in KV cache management across block sizes as a function of sequence length, dropping from 11.81% at 4 KB to 0.00% at 256 B.

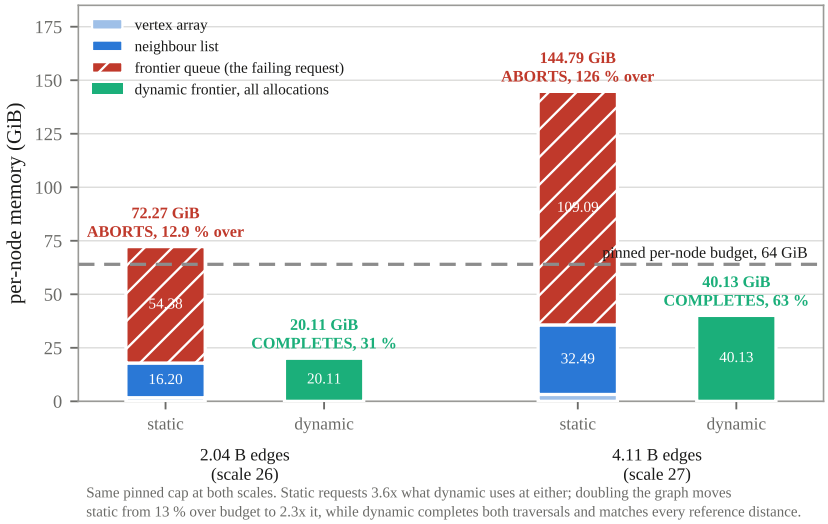


Figure 5.30: Physical memory footprint on scale-26 BFS traversal. Static worst-case reservation exceeds the 64 GB budget (112.9%) and aborts, whereas GLMalloc completes within 31.4% of available memory.

Backend Configuration with Application Knowledge

LLM Serving (`llm_serve`) In LLM KV cache management workloads, allocation accounts for 45.1%-48.3% of the serving loop's instructions. As shown in Figures 5.26-5.28, sweeping block sizes from 4 KB down to 256 B causes allocator per-call latency to vary by only 12.8% (397.8 to 448.8 cycles on slab, Figure 5.26), while internal memory fragmentation drops monotonically from 11.81% at 4 KB to 0.00% at 256 B (Figure 5.29). The end-to-end token processing time (Figure 5.28) shows that 512 B blocks achieve 22% faster token generation than 4 KB blocks by eliminating the wasted KV data paddings transferred across memories. Throughput peaks at 8.56 allocations/cycle at 82% occupancy (Figure 5.27) due to cache capacity and queue contention.

Dynamic-Frontier BFS (`bfs_dynarray`) In push-style BFS traversal, frontier queue sizes fluctuate unpredictably across levels. Static worst-case reservation needs to provision for maximum queue boundaries simultaneously (∼28.5 bytes per edge). Figure 5.30 compares static reservation against GLMalloc dynamic allocation on a graph of 28.29 mil-

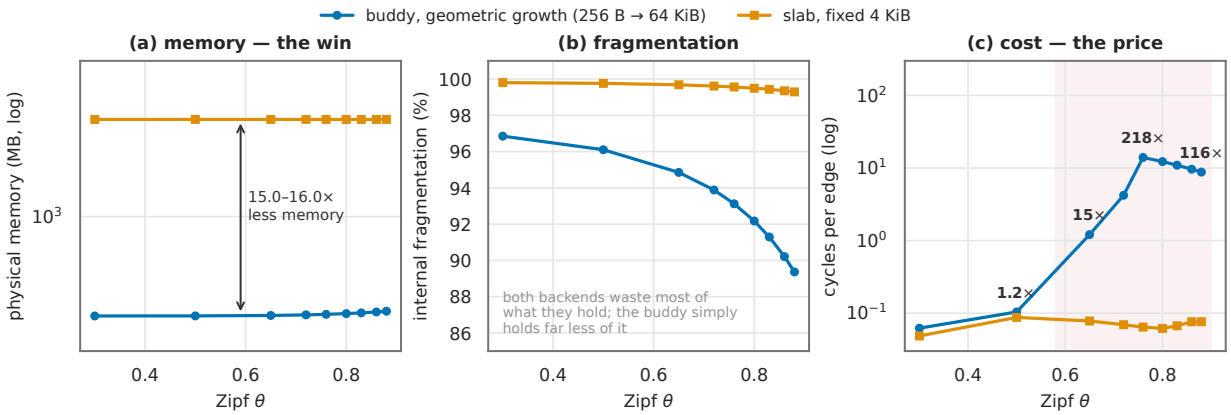


Figure 5.31: Allocator backend trade-off on `graph_ingest`

lion vertices and 2.04 billion edges under a pinned 64 GB per-node budget. Static reservation requires 72.27 GB (112.9% of budget), causing an immediate out-of-memory abort. In contrast, GLMalloc dynamically allocates frontier blocks per level and recycles completed frontier memory after each phase, executing the full six-level search within a peak footprint of 20.11 GB (31.4% of available memory) with 100% distance verification. Dynamic memory management enables the computation to execute within physical memory limits.

Graph Ingestion (`graph_ingest`) Figure 5.31 evaluates whether different allocator backend choices affect the performance on `graph_ingest`. In irregular graph analytics, power-law vertex degree distributions cause adjacency list lengths to vary by orders of magnitude, e.g., billions of degree-1 vertices in contrast to a few vertices with billions of neighbors. Under the Fixed-Size Slab backend, every neighbor list occupies at least one 4 KB block, causing degree-1-dominated graphs to suffer 99.3%-99.8% internal fragmentation and consume an additional 4 GB of memory.

In contrast, the Multi-Order Buddy Vector backend with geometric growth (256 B base block up to 64 KB, operating in DRAM-backed mode for large graphs) serves the identical power-law edge stream in 246.2 MB to 263.2 MB across $\theta \in [0.30, 0.88]$, a 15.0× to 16.0× reduction of memory usage.

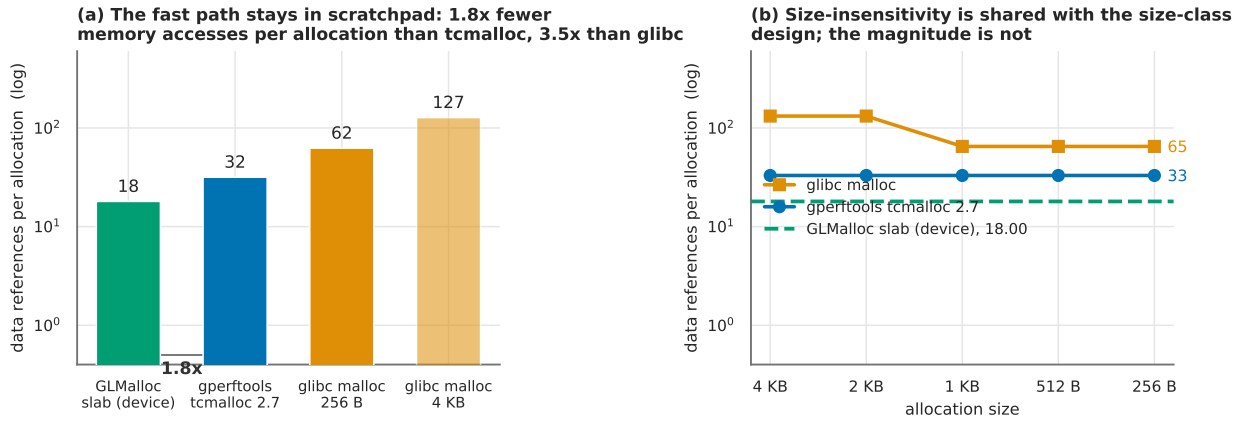


Figure 5.32: Microarchitecturally invariant execution costs across memory allocators on an unloaded fast path. GLMalloc executes in 73 instructions and 18 private scratchpad references with zero DRAM accesses and zero atomic operations, compared to 31.7–127 references for production CPU host allocators.

Nevertheless, this memory footprint reduction results in a price in execution time: while per-edge cost is within 1.3× of the slab through $\theta = 0.50$, as the graph becomes more skewed, the imbalanced degree distribution results in grown blocks for a few high-degree vertices, increasing buddy per-edge cost by 15× at $\theta = 0.65$ and peaking at 219× at $\theta = 0.76$. Backend selection is a performance-vs-memory trade-off exposed to the software: the buddy backend is required to fit the stream into physical memory, while the slab backend is one to two orders of magnitude faster when memory is unconstrained.

GLMalloc gives applications the flexibility to choose and configure the backend allocator while keeping the memory layout the same to maintain locality. All the measurements above keep the allocations local to the node, i.e., the node-local Level-One configuration in Section 5.3.1.

5.3.4 Comparison to Existing Allocators

In this section, we position GLMalloc against a list of existing allocators. We compare GLMalloc against established production CPU allocators (glibc `malloc` (ptmalloc2-derived) [43], Google `tcmalloc` [125], Microsoft `mimalloc` [70], and `jemalloc` [34]) and near-memory archi-

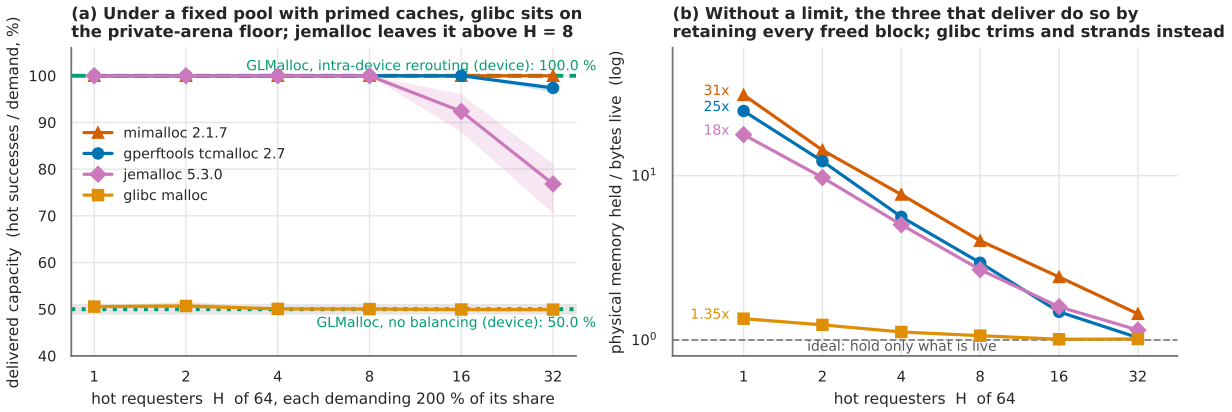


Figure 5.33: Delivered capacity under spatial demand skew across allocators.

tectures (`PIM-malloc` [69]). Because clock frequencies and ISAs differ substantially between host processors and accelerator lanes in our evaluation system, we anchor the comparison in microarchitecture-invariant dimensions: memory hierarchy penetration, atomic instruction density, memory reference footprint, and capacity delivery under adversarial skew.

Memory Reference Density

As shown in Figure 5.32, GLMalloc's slab fast path executes in 73 instructions and generates 18 private scratchpad SRAM references. Including requester-side overhead yields 100 instructions and 1 DRAM store per allocation. In contrast, production allocators execute between 31.7 memory references (`tcmalloc`) and 62-127 references (`glibc`). When an allocation misses thread-local caches, allocators traverse complex pointer-linked free lists or page-to-span radix trees under mutex locks, whereas GLMalloc lanes exclusively own their local allocation bitmaps, eliminating all expensive atomic operations.

Adversarial Demand Skew: Capacity Recovery, Reservation Headroom, and Memory Bloat

Under spatial demand skew (Figure 5.33), production allocators' performance is limited:

- **glibc malloc** and **PIM-malloc** assign threads to fixed arenas or cores, stranding capacity in idle arenas and delivering only 50% of requested capacity.
- **tcmalloc** and **mimalloc** achieve 100% delivered capacity, but do so by dynamically expanding private heaps from the global virtual address space, retaining 12.2× to 31.0× more memory than live object volume when freed memory is cached thread-locally.
- **GLMalloc** achieves 100% delivered capacity within a fixed physical memory cap via a 3-tier load-balancing mechanism.

We distinguish between dynamic bloat and static memory reservation to better understand the memory footprint:

1. **Dynamic Memory Bloat (Conventional Allocators)**: In TCMalloc and mimalloc, memory bloat represents the expansion of the process's virtual address space. Because active threads expand private heaps from the OS without returning freed blocks across thread boundaries, memory held over live bytes can escalate by many times, potentially leading to out-of-memory crashes.
2. **Static Reservation (GLMalloc)**: In contrast, GLMalloc operates without demand paging because centralized memory allocation and privileged address (re-)mapping are expensive in scale-out systems. Level-One segments are allocated once at initialization. When measured on the host allocators' metric (total physical reservation divided by active live bytes), GLMalloc exhibits an $\propto 1/H$ geometric curve ($\approx 32\times$ at $H = 1$ down to $1.0\times$ at $H = 32$).

The distinction of GLMalloc is that it achieves 100.0% delivered capacity under adversarial skew without expanding its memory reservation. By redistributing capacity across private sub-pools via Tier 1 intra-device rotated rerouting and Tier 2 inter-device region redistribution, GLMalloc recovers the full capacity within a deterministic memory capacity ceiling.

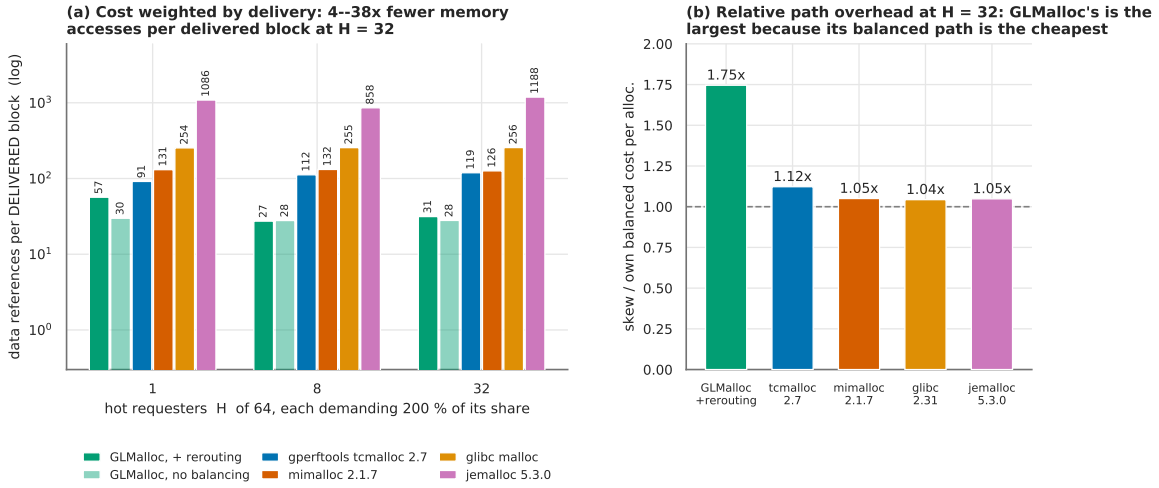


Figure 5.34: Weighted execution cost under spatial demand skew.

Delivery-Weighted Execution Cost

Figure 5.34 evaluates the execution cost for memory allocation. Under lock contention and failure retries, host allocators spend 119-256 references per delivered block at $H = 32$. GLMalloc spends 18 references when balanced and 31.4 under skew, achieving 3.8× to 8.2× higher efficiency.

5.4 Summary

GLMalloc provides a scalable, hierarchical memory management scheme co-designed with the UDTrans hardware translation scheme. By decoupling privileged segment partitioning from unprivileged, decentralized local allocation, GLMalloc resolves the tension between coarse-grained hardware address translation and fine-grained dynamic allocation.

Evaluation across synthetic microbenchmarks, irregular graph workloads, and LLM serving demonstrates that GLMalloc delivers:

- **Constant Fast-Path Cost and Amortized Privilege Overhead**: low latency with no atomic operations or DRAM metadata accesses, and privileged segment setup amortized over 10^5–10^7 unprivileged allocations during the application runtime.

- **2-Dimensional Address Space Locality Control**: Elimination of cross-node memory stores (75% $\rightarrow$ 0%) by configuring swizzle parameters, delivering a 1.42×–1.45× barrier speedup.

- **Skew Resilience and Scalability**: Recovering 100% capacity while delivering under spatial demand skew at 29%–36% lower cycle latency, handling whole-node oversubscription below 100% global utilization via Tier 3 request forwarding, and robust tolerance of dynamic allocate-and-free, governed by an analytical threshold ($\sigma_{\text{lane}} > 1/\rho$).

- **Real-world Performance**: 98% weak-scaling efficiency up to 131,072 lanes, saving 15×–16× footprint through backend selection, completing dynamic BFS within 31.4% memory where static reservation aborts, and comparing against production host allocators with architectural invariant metrics.

CHAPTER 6

DISCUSSION AND RELATED WORK

In this chapter, we discuss the implications of UDMem, review prior work in distributed systems and memory architecture, and position our work within this literature.

6.1 Discussion: Implications for Future Global Memory Architecture, System Software, and Runtime

The findings in Chapters 4 and 5 carry broad implications for the design of distributed memory architectures, operating system control planes, and runtime software in next-generation accelerator systems, e.g., the 1024-node integrated systems that the Nvidia NVL and Google TPU systems are evolving into. We divide them into the hardware-facing and software-facing dimensions and discuss them in detail below.

6.1.1 Implications for Global Memory Architecture and Hardware Co-Design

Rethinking the Virtual Memory Abstraction

Traditional computing systems treat virtual memory as a monolithic abstraction that couples physical page backing, access protection, address translation, and dynamic memory allocation into uniform multi-level page-table radix trees. At the scale of petabyte distributed systems containing hundreds of thousands of cores, this monolithic scheme collapses: maintaining fine-grained 4 KB or 2 MB page descriptors exhausts TLB capacity, triggers substantial page-table walking penalties, and generates expensive distributed TLB invalidation. GLMalloc and UDTrans decouple privileged translation from fine-grained management. We show that coarse-grained, 2-dimensional hardware segment translation provides memory safety, address virtualization, and inter-process isolation with minimal translation

cost, while unprivileged, decentralized software runtimes can handle dynamic fine-grained allocation at high throughput. In other words, hardware translation enforces coarse-grained address boundaries and safety invariants; runtime software governs fine-grained allocation lifetimes and fragmentation. We believe this modular structure is a scalable way forward.

Programmable Translation for Locality Control

Historically, hardware address translation has operated as a passive mapping layer invisible to application software. GLMalloc demonstrates that exposing hardware swizzle translation parameters (B and C) transforms address translation into an active locality control plane. By exposing swizzle granularity directly to unprivileged runtime software, the runtime can align memory allocation with hardware striping bounds, eliminating cross-node memory stores without altering application source code, inserting software indirection tables, or modifying hardware translation entries. Eliminating remote memory accesses produces a substantial speedup (1.42×-1.45×) across the workloads running on distributed systems. Future distributed memory architectures should expose configurable translation parameters to user space as first-class architectural primitives.

Private Spatial Allocation Arenas across Memory Hierarchies

While GLMalloc is evaluated on the UpDown architecture with private software-managed scratchpads, its core design principles transfer directly to emerging cache-coherent shared-memory architectures (e.g., CXL 3.0 fabrics, NVLink-connected multi-GPU clusters, and unified-memory APUs). In any system featuring massive thread concurrency, sharing allocator metadata or centralized global free lists across cores is costly: severe cache-line thrashing, false sharing, and bus invalidation storms. Structuring memory management into private arenas/pools with decentralized, message-based rebalancing can organize memory access to avoid coherence thrashing across the memory hierarchy.

6.1.2 Implications for Operating Systems and Memory Management

Lightweight Multi-Tenancy and Context Switching

In virtualized cloud environments and shared accelerator clusters, multi-tenancy typically incurs severe performance degradation due to page-table walking depth, nested virtualization page faults, and cross-core TLB shootdowns. Under UDTrans's 2D swizzle segment architecture, each tenant or kernel context requires only 2 to 6 segment descriptors. Therefore, context-switching an execution lane or tenant requires swapping an 8-to-16-entry descriptor table, eliminating deep page-table walks, memory-footprint bloat, and distributed TLB shootdown storms that paralyze multi-tenant paged architectures. We believe UDTrans is an attractive solution to these problems.

Scalability of Privileged Operations

In petabyte-scale accelerators, crossing the privileged operating system boundary (e.g., host driver calls, hypervisor traps, global manager locks) is orders of magnitude more expensive than local processor arithmetic. GLMalloc demonstrates that operating systems and accelerator control planes should manage memory at coarse granularities. Privileged Level-One management is invoked once per aggregate data structure during initialization, configuring base bounds and swizzle bitmasks across participating nodes. Because a single Level-One segment amortizes over 10^5 to 10^7 unprivileged allocations, the privileged control plane remains decoupled from dynamic allocation volume and thread concurrency, ensuring that operating system overheads do not scale with application execution length. In short, GLMalloc is an attractive approach for scaling.

Allocator Balancing vs. Application Sharding Compilers and algorithms strive to distribute graph vertices and tensor shards evenly across distributed memory. However, data-dependent irregular workloads typically produce dynamic, unpredictable runtime skew

that static analysis cannot anticipate. GLMalloc demonstrates that allocator load balancing can be an essential runtime safety net: application placement optimizes fast-path efficiency, while hierarchical allocator load balancing guarantees forward progress and converts stranded capacity under adversarial runtime skew without job abortion. GLMalloc provides scalable, flexible management of locality.

6.2 Related Work

This dissertation fills an important gap by showing that fast, flexible memory management can be achieved in a scale-out distributed system. First, conventional address translation schemes cannot provide unlimited translation reach, sub-linear metadata scaling, and fine-grained, programmer-directed physical placement across distributed memories. Second, state-of-the-art memory management runtimes do not combine explicit cross-node physical placement mechanisms with dynamic, skew-resilient, fine-grained allocation beneath that placement abstraction. In the following sections, we systematically review the literature and discuss how it relates to our approach. We divide prior art into four research domains, each addressing one aspect of the design space that UDMem targets.

6.2.1 Address Translation: Page Versus Segment Paradigms

Scalable address translation for shared memory remains a bottleneck in shared memory systems and is amplified by non-uniform memory access (NUMA) and symmetric multiprocessing (SMP) [68, 111]. These architectures often adopt hierarchical radix-tree page tables that operate at a nominal 4 KB granularity for address translation. A wide range of optimizations have been proposed to reduce radix-tree traversal latency: intermediate page-walk caching [11], dedicated long-reach MMU caches [18], low-latency traversal schemes [93], translation prefetching [76], and recency-based preloading [104]. A complementary line of work tries to increase translation reach within fixed-granule constraints by coalescing adja-

cent page mappings in hardware [95], extending coalesced descriptors across physical page migrations in heterogeneous memories [115], or leveraging spatial subregion contiguity [122].

Although these strategies mitigate translation latency, they fail to address the fundamental limitation that prevents scaling: the underlying physical frame remains the atomic unit of translation, and metadata overhead scales with mapped address capacity ($\mathcal{O}(N)$). Therefore, expanding TLB reach is achieved by coarsening the granularity of translation. Huge pages are the most common structural optimizations for increasing granularity [27, 28]. Another proposal, Ingens, addresses this trade-off by refining the runtime promotion policy for huge-page construction rather than modifying the underlying page-table mechanics to dynamically mitigate allocation overhead [65]. Nevertheless, none of the page-based proposals eliminates the explosion of translation state as the dataset size increases to terabytes or petabytes, unlike UDTrans and other segment-based translation approaches.

Segment and range translation schemes eliminate the limitation entirely, providing unlimited TLB reach via segment-based translation entry. Direct Segments dedicate a single base-limit register to bound translation state to $\mathcal{O}(1)$ complexity for memory-intensive applications [13]: a bulk region is allocated at initialization. Redundant Memory Mappings generalize this abstraction to multiple independently sized virtual ranges per process [57]. FlexPointer embeds hardware range comparators and tagged pointers directly into the processor translation pipeline [24], and direct segment translation has been successfully synthesized in physical silicon [63]. Midgard addresses translation overhead by re-architecting the memory access hierarchy rather than adjusting mapping granularity [50].

Nevertheless, existing segment-based schemes exclusively target single-node shared-memory topologies and fail to support variable-sized, node-level data distribution across disjoint physical nodes that parallel applications need for bandwidth and locality. UDTrans addresses this limitation: each entry encodes the virtual base, physical base, and contiguous size, plus a swizzle mask to express non-contiguous multi-node affinity and/or block-cyclic data

distribution, which are common in mainstream parallel applications.

6.2.2 Hardware Address Distribution in Classic Distributed-Memory Machines

Hardware support for scattering address ranges across memory nodes existed in the early history of distributed-memory machines. For example, in Cray T3D and T3E, the software can supply a mask per data structure, expressing data distributions over physically disjoint memories, so that consecutive indices of a shared array land on successive PEs in a blocked, cyclic, or block-cyclic pattern by hardware [59, 2, 106, 107]. Later frameworks like CRAFT, SHMEM, and Co-Array Fortran elaborate this mechanism [84]. Similar proposals also exist in IBM's RP3, whose pages could be marked as interleaved-across-modules [94], and in the Tera MTA and Cray XMT, whose addresses can be scattered across memory units to defeat hot-spotting [8, 36].

UDTrans's swizzle translation builds on this idea and tailors for future accelerator-based distributed systems. In the T3E, address distribution and protection were separate steps in the E-register path and memory references required managing the E-registers explicitly, whereas UDTrans fuses translation and protection into a single swizzle translation descriptor (Figure 4.5), so a UD-TLB hit resolves both in 1 cycle on the ordinary access path(Sections 3.2.1 and 4.1.5). In addition, the mask parameters (B, C) are set by a privileged runtime at allocation time and used as a locality knob (Chapter 5), rather than fixed by a compiler at declaration.

6.2.3 RDMA, CXL, and Disaggregated Memory Systems

Remote Direct Memory Access (RDMA) is widely adopted for high-throughput, low-latency inter-process communication, enabling direct memory access across network fabrics without kernel involvement [119]. Despite the wide adoption, hardware memory registration incurs

substantial runtime overhead [38]. Furthermore, the runtimes are forced to amortize memory pinning and registration costs with large, coarse memory pools instead of dynamic, fine-grained mappings, because of the limited registered memory translation tables [19]

Building on RDMA, prior work abstracts multi-node memory into unified global address spaces: FaRM guarantees distributed transaction serializability [32], Grappa conceals network latency through massive lightweight-thread parallelism without enforcing spatial placement optimizations [82], and GAM integrates a cache-coherence protocol to permit remote page caching across compute nodes [20].

Although these frameworks show that unified global address spaces across distributed clusters are feasible, they fail to scale because they lack native address-translation mechanisms. RDMA operations depend on the host operating system's page tables and I/O Memory Management Units (IOMMUs) for address translation, so they inherit the same translation-reach and TLB bottlenecks as shared-memory systems. Furthermore, data locality in these systems remains a result of execution scheduling rather than controlled placement policy: FaRM incorporates a standard slab/region allocator, Grappa operates without a fine-grained allocation engine, and GAM enforces caching exclusively at page or coarse-region granularity.

Compute Express Link (CXL) memory pooling provides cache-coherent memory expansion and pooling at datacenter scale, but at coarse granularity. Pond provides a framework for orchestrating CXL memory pools to minimize cloud DRAM fragmentation, partitioning capacity across physical sockets [74]; TPP executes transparent page migrations between local DRAM and disaggregated CXL memory tiers based on access frequency [78]; DirectCXL implements load/store semantics that bypass software paths [47]; and recent surveys formalize the broader architectural challenges of memory disaggregation [35, 77].

None of these disaggregated architectures decouples translation reach from physical placement granularity. Furthermore, direct physical addressing across disaggregated nodes is re-

stricted to local rack clusters. Like RDMA, CXL memory extensions expand raw addressable capacity while leaving sub-page spatial distribution unaddressed.

UDMem's two-dimensional address translation and hierarchical management provide a natural architectural foundation for CXL memory fabrics:

- By representing CXL device IDs and port coordinates as the node dimension in a 2D physical address space, hardware swizzle translation can distribute striped allocations across multi-device CXL memory pools without incurring host-side page-table walk traffic over PCIe/CXL links.
- The two-level control plane allows host operating systems to assign coarse CXL memory extents to accelerator contexts, while unprivileged accelerator runtimes manage fine-grained, low-latency object allocation locally, bypassing host OS driver bottlenecks entirely.

6.2.4 Scalable Memory Management

Single-Node High-Throughput Allocation

Modern high-performance memory allocators achieve scalability by eliminating shared synchronization primitives. Hoard introduced a modern thread-local heap abstraction, backed by a centralized global pool, that provides provable theoretical bounds on external-memory fragmentation while keeping thread-local allocation paths synchronization-free [17]. jemalloc utilizes static round-robin thread-to-arena assignments paired with size-class bins to minimize lock contention and fragmentation concurrently [34]; mimalloc partitions free lists on a per-page basis within local heaps, optimizing CPU cache locality through lock-free structures [70]; and Streamflow couples contention-free local allocation with an asynchronous remote-free protocol across per-thread arenas, decoupling cross-thread deallocation from the allocating thread's critical path [105]. TCMalloc scales these techniques to hyperscale pro-

duction deployments [125], and comparative evaluations systematically characterize these design paradigms [37].

These allocators demonstrate that partitioned, arena-local architectures are essential for low-latency, fine-grained dynamic allocation on single compute nodes. However, they do not resolve the challenges of distributed, scale-out memory hierarchies. Each allocator assumes a uniform, single-node physical address space: Hoard relies on a centralized global heap for balancing, jemalloc maintains static, NUMA-oblivious arena assignments without dynamic cross-node rebalancing, and mimalloc limits locality optimizations to intra-node, page-level sharding. While Streamflow's remote-free mechanism set a precedent for asynchronous cross-arena reclamation, its scope remains strictly bounded by single-machine physical memory.

These architectures do not perform multi-node placement decisions because their execution model assumes uniform topologies. Scalable dynamic allocators targeting constant-time $\mathcal{O}(1)$ allocation latency [110] operate under identical single-node assumptions. In contrast, GLMalloc uses 2-dimensional address spaces for allocation and provides applications a simple interface to choose between maximizing local access and distributing for bandwidth.

Operating-System NUMA Allocation Mechanisms

Modern operating systems on NUMA architectures default to a first-touch allocation policy [67]. Interfaces including `numactl`, `libnuma`, and the low-level `mbind` and `set_mempolicy` system calls provide explicit user-level placement control across processes, memory ranges, and allocations [61]; kernel-level automatic NUMA balancing executes background page migrations guided by periodic unmapping and page faults [46]; and specialized frameworks distribute page-table structures and TLB shootdown domains across NUMA topologies [41].

Lepers et al. show that simple NUMA heuristics, such as uniform local thread-to-node co-location, often fail to maximize efficiency, because network and memory bandwidth saturation frequently occurs [71]. This matches the findings in Section 4.3.3: first-touch allocation

can match UDTrans locality only when target memory affinity aligns with page boundaries or coarser granularities. Therefore, the limitations documented in this dissertation arise from the inability to express, enforce, or manage sub-page placement in traditional NUMA systems.

Programmer-Controlled Placement in PGAS Runtimes

Partitioned Global Address Space (PGAS) models enable programmers to define data distribution by runtime systems: UPC exposes parameterized cyclic and block distributions [42], Chapel provides `Block` and `Cyclic` domain maps [21], OpenSHMEM defines symmetric memory heaps with direct one-sided put/get semantics [15], Global Arrays enforces explicit distribution geometries upon array instantiation [83], X10 models spatial execution domains as first-class places [23], and UPC++ provides global pointer templates that encapsulate distributed node ownership semantics [10]. Chapel domain maps further generalize this paradigm, allowing distribution topologies to be implemented as modular software libraries rather than static compiler built-ins [22]. Similarly, the Legion programming model decouples application data declarations into logical regions so physical placement is expressed by a software-defined **mapper** [14].

Programmer-controlled data distribution is well studied in the PGAS world. However, PGAS runtimes lack the underlying translation mechanisms evaluated in Section 4.3: a compact hardware translation mechanism that understands declarative distribution policies and translates them into hardware addresses on every memory access. In PGAS runtimes, layout directives are processed in software via general-purpose instructions during runtime initialization or communication phases. Moreover, conventional hardware segment translation would incur $\Theta(\#\text{allocations})$ state complexity if applied to every dynamic allocation; UDTrans and GLMalloc overcome this by bounding hardware translation to $\mathcal{O}(D)$ state for D coarse data structures while delegating fine-grained dynamic allocations to unprivileged

software.

Furthermore, PGAS runtime manages placement at the granularity of coarse arrays or task-level regions. They provide no mechanisms to mitigate allocation skew when a specific locale's memory partition is exhausted by fine-grained allocations. While Legion's mapper deliberates asynchronously over coarse data dependencies, GLMalloc Level-Two's load balancing mechanisms resolve imbalanced fine-grained allocation on the critical execution path.

6.2.5 Accelerator and Domain-Specific Global Memory Systems

Modern scalable compute clusters also suffer from address-translation overhead due to their specialized memory hierarchies. The exponential growth of ML model parameters and datasets exacerbates the issue: GPU and TPU High-Bandwidth Memory (HBM) are capacity-bounded. While Unified Virtual Memory (UVM) in GPU systems enables transparent oversubscription into host DRAM, it incurs severe PCIe and NVLink overhead [5]. Prior work evaluates these performance bottlenecks and analyzes the page-fault processing, page-migration overheads, and thrashing across production workloads [6], at sub-page granularity [4], across shared virtual memory implementations [29], and through standardized benchmarking suites [112]. GPU demand paging introduces substantial latency during page faults [124]; batched faults serialize the execution, degrading throughput in irregular, sparse-access workloads [60]; and the hardware prefetching and eviction heuristics exhibit thrashing under oversubscription [40, 62].

These UVM mechanisms operate through host-side kernel drivers that orchestrate discrete host-device pairs. Extending this model across multi-GPU architectures preserves the underlying page-based translation. Prior multi-GPU research investigates TLB entry sharing and spilling across devices [73], hybrid hardware-software NUMA management [121], optimized page-migration runtimes [12], and reduced TLB shootdown overheads [72], alongside

architectures integrating full x86-64 translation engines into wide SIMD pipelines [96] and extending TLB reach for accelerated processors [54].

Nonetheless, irregular, data-dependent memory access patterns across distributed accelerators remain an unresolved failure mode [58]. Because these GPU systems still use fixed-size page frames, they face the same reach-versus-placement trade-offs as conventional page-based CPU systems. The same bottleneck we showed in Chapter 4 for 4KB and 2MB pages carries over to Nvidia GPU clusters, so as memory footprints and machine scales keep growing, performance will be limited by the coupling between reach and placement. Whereas UDMem provides a cheap scalable solution via swizzle translation in UDTrans.

Tensor Processing Units (TPUs) manage data orchestration via static compiler scheduling, removing demand address translation and runtime TLB structures from the execution path [56, 55, 44]. Deterministic dataflow engines [1], reconfigurable spatial architectures with high-density on-package memory [97], and specialized domain accelerators [51] use similar static scheduling to eliminate runtime management overhead. This works when memory access patterns are determinable at compile time, but fails to support irregular, input-dependent access patterns such as the dynamic workloads characterized in Section 2.2.2.

In addition, application processes are statically scheduled across TPUs and spatially share compute and memory resources in the system. Not only do they lack a global address-space abstraction, but they also cannot safely support multi-tenancy and memory pooling between applications for dynamic runtime memory management. UDMem, in contrast, provides a safely isolated address space for each application when they spatially and temporally share the system, and lets applications share data via memory pooling or pipes, similar to how processes would share in a conventional single-node shared-memory system.

Advanced rack-scale coherent fabrics and interconnects, like NVLink networks, construct a unified memory domain from distributed memory nodes for the application but do not expose fine-grained placement to software [89, 87, 86]. Evaluations on production systems

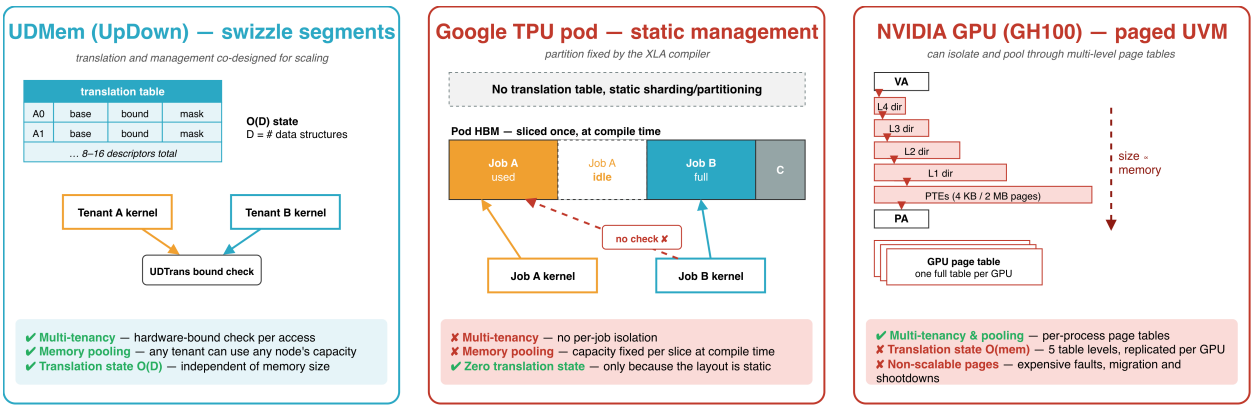


Figure 6.1: Comparison between GPU Supercomputer, TPU SuperPod, and UDMem global memory machine

show that inter-node data movement still dominates execution time [39]. Hardware cache coherence masks the physical location of data, but it does not provide software abstractions to direct or enforce intentional physical data placement. UDTrans, by contrast, addresses this limitation directly with programmer-controlled swizzle masks. Figure 6.1 gives a cross comparison between the global memory systems that UDMem enables and existing ones.

Memory Allocation in Heterogeneous Systems

GPU Dynamic Allocators Dynamic allocation for massively paralleled concurrent threads has been studied intensively by GPUs [52]; ScatterAlloc hashes threads onto superblocks and pages to scatter contention and reports roughly 100× the throughput of the CUDA toolkit allocator [109]; Halloc combines size-class slabs with hashed bit-array search and exceeds 10^9 malloc/free pairs per second for small objects [3]; DynaSOAr specializes allocation for structure-of-arrays object layouts [108]; Ouroboros uses virtual queues to keep track of freed memory and reuse across size classes to reduce fragmentation [117]; and Gallatin builds a general-purpose vairable sized allocator on a concurrent tree [79]. Winter et al. benchmark these designs systematically [118].

These allocators share several design principles with GLMalloc: private arenas/pools

without lock contention, bitmap metadata, slab or buddy backends. However, none of the GPU allocators are designed for scale-out systems like in UDMem. They view memory as a flat address space because GPU memory lacks the abstraction UDMem uses to express distributed memory topology.

Near-Memory Accelerator Allocators Although near-memory computing is an emerging technology, researchers have proposed preliminary allocator designs for accelerator-based systems such as Process-In-Memory (PIM) and Processing-Near-Memory (PNM). These architectures integrate lightweight compute cores directly adjacent to DRAM banks.

PIM-malloc provides a high-throughput dynamic memory allocator for the UPMEM-PIM architecture where each DPU core is bound to an isolated, bank-level DRAM segment [69]. The design focuses on optimizing local metadata organization and the execution overhead of allocation routines, without providing inter-core request routing, load rebalancing, or remote bank allocation. Its allocation model remains solely node-local, limited by the underlying hardware constraints.

PUMA explores a complementary design point as a placement-aware allocator for processing-using-DRAM architectures [90]. To satisfy the operational requirements of in-DRAM bitwise computation, PUMA ensures that interacting operands are co-located within identical memory subarrays and are strictly row-aligned. It achieves this by partitioning huge pages into fine-grained, page-aligned segments using reverse-engineered internal DRAM topology maps. Thus, PUMA explicitly manages physical placement relative to memory hardware structures rather than treating placement as incidental. However, PUMA targets subarray-level alignment constraints within an individual memory module, lacking support for multi-node network topologies, hierarchical interconnects, or dynamic cross-node load rebalancing.

None of the accelerator-based allocators solve the dual challenges UDMem addresses: topology-aware physical placement across distributed memories, with dynamic, skew-resilient fine-grained allocation on top. The key reason is that existing PIM systems are restricted by

isolated address spaces: each PIM core can access only its local DRAM bank, and hardware does not support cross-core or cross-node memory placement.

The UDTrans and GLMalloc co-design provides a blueprint for general-purpose, scale-out PIM systems:

- UDTrans's reversible swizzle logic requires negligible silicon area and power (under 17 mW per unit), making it feasible to embed directly into near-memory logic dies.
- GLMalloc's scratchpad-resident metadata structures eliminate DRAM traffic from the allocation path, ensuring that near-memory compute cores do not pollute scarce DRAM bank bandwidth with allocation bookkeeping.

CHAPTER 7

SUMMARY AND FUTURE DIRECTIONS

This dissertation has addressed the core memory management challenges in massive-scale distributed architectures: 1) the address-translation bottleneck caused by scaling virtual memory over petabytes of distributed physical memory, and 2) the memory-management challenge of serving millions of concurrent, dynamic allocations with global privilege control. This chapter summarizes our core findings and contributions, examines the implications for emerging hardware and software, and outlines directions for future research.

7.1 Insights for Global Memory Systems

The findings in Chapters 4 and 5 demonstrate two foundational principles that guided the UDMem design.

7.1.1 Decoupling Translation Reach Granularity from Data Placement

A central lesson of this dissertation is that coupling the translation granularity to the placement granularity is not ideal for scale-out memory systems.

In conventional virtual memory architectures, the page is the unit of translation, physical memory allocation, and data placement. Therefore, increasing page size (e.g., from 4 KB to 2 MB or 1 GB huge pages) to expand TLB reach inevitably coarsens the granularity at which data can be distributed across physical nodes. In a distributed global memory system, coarsening the placement unit destroys the programmer's ability to interleave data across nodes for bandwidth or align partitions with local compute units. In contrast, maintaining fine-grained placement at 4 KB granularity forces translation state to scale as $\mathcal{O}(\text{pages})$, resulting in TLB reach thrashing and expensive page walks.

UDTrans demonstrates that translation reach and placement are orthogonal dimensions that can be cleanly separated in hardware:

- **Reach** is governed by variable-sized segment base-and-bound registers, providing unbounded reach per descriptor ($\mathcal{O}(1)$ state per data structure).
- **Placement** is governed by bit-level swizzle masks (B, C), enabling fine-grained, power-of-two block-cyclic interleaving across distributed nodes without requiring additional descriptors.

This separation preserves single-cycle translation cache hits regardless of dataset footprint while granting software fine-grained control over physical data layout.

7.1.2 Decoupling Privileged Protection from Unprivileged Execution

Another principle governing scale-out software memory management is to separate privileged global access mediation from the runtime allocation path.

In traditional operating systems, every dynamic allocation that maps virtual memory invokes privileged kernel services (e.g., `mmap` and/or page table manipulation). In an accelerator system with thousands of millions of concurrent execution lanes, requiring privileged intervention or centralized metadata synchronization on dynamic allocations causes severe delays in the critical path.

GLMalloc establishes that system-wide protection and dynamic memory management should be partitioned and managed in a hierarchical scheme:

- **Privileged Physical Partitioning (Level-One)**: A centralized, privileged host manager (`DRAMalloc`) executes off the critical path, partitioning physical memory and controlling hardware translation descriptors. Because this privileged grant is coarse, the application runtime amortizes its cost.

- **Unprivileged Decentralized Allocation (Level-Two)**: Unprivileged, lane-local allocators (`MemoryPool`) execute directly on accelerator lanes, servicing fine-grained allocations out of lane-private scratchpad memory with zero DRAM accesses and zero atomic operations.

By ensuring that hardware translation entries enforce process isolation while unprivileged software manages allocation within the segment boundaries, the system achieves isolation and safety without sacrificing dynamic fine-grained management throughput.

7.2 Open Problems and Future Research Directions

While this dissertation demonstrates the scalability and efficiency of co-designed address translation and memory management, several questions remain open for future exploration.

7.2.1 Dynamic Segment Remapping

In the current GLMalloc design, a Level-One swizzle segment is allocated with a specific configuration. In long-running, multi-phase execution pipelines, distinct application phases may have contrasting allocation demands: an initial phase may require highly local buffer access, followed by a phase dominated by billions of random accesses across the structure that demand global bandwidth.

Transitioning between allocation mappings requires tearing down the Level-One segment and establishing a new segment reservation. This teardown incurs global synchronization, TLB entry invalidations, and translation table re-registration across all participating nodes. An important open research direction is developing unprivileged segment remapping that allows applications to reformat and remap their segments locally into alternative-size interleaved segments within a physical segment.

7.2.2 Scratchpad-Resident Multi-Class Geometric Size Bins

GLMalloc supports both uniform Fixed-Size Slabs and power-of-two Buddy. However, the buddy backend bounds worst-case internal fragmentation at 50% (since each request is rounded up to the next power of two). In contrast, commercial allocators such as `jemalloc` and `tcmalloc` maintain geometric size classes (typically 4 or 8 discrete size bins per power-of-two doubling), reducing worst-case internal fragmentation to $\leq 12.5\%$.

In accelerator architectures with strict scratchpad area limits, implementing fine-grained geometric sizes presents the trade-off between metadata storage and instruction latency. Future work should investigate compressed, scratchpad-resident bitmap hierarchies that manage geometric size classes while preserving the low allocation latency.

7.2.3 Multi-Dimensional and Non-Power-of-Two Swizzle Descriptors

The current UDTrans swizzle descriptor encodes block-cyclic striping using power-of-two parameters: block size 2^C and node cardinality 2^B. While power-of-two block-cyclic interleaving matches the standard distribution directives of MPI and PGAS frameworks (e.g., UPC, Chapel, and OpenSHMEM [7, 21, 10, 15]), there exist workloads that exhibit non-power-of-two or multi-dimensional decomposition patterns:

- **Non-Power-of-Two Node Counts**: Systems deployed with node counts that are not pure powers-of-two (e.g., 384 nodes or 3-chassis configurations) cannot utilize the single-mask swizzle translation without padding physical address space.
- **Multi-Dimensional Layouts**: Workloads such as the Needleman-Wunsch (NW) dynamic programming algorithm or 2D blocked matrix tiling traverse data along diagonals or 2D sub-grids, which cannot be expressed via a single one-dimensional bit-interleave mask.

Extending UDTrans to support multi-dimensional affine transformations or reversible non-

power-of-two modular mappings while retaining single-cycle hardware evaluation and reversibility represents a promising area of hardware research.

7.2.4 Dynamic Interleave Adaptation

The swizzle locality evaluation in Chapter 5 demonstrated that configuring swizzle parameters (B, C) to eliminate cross-node memory stores delivers substantial barrier speedups.

In workloads with massive streaming collectives or dense all-to-all tensor exchanges, memory controllers and network links operate near bandwidth saturation. Under such conditions, concentrating write traffic on local memory nodes can cause localized memory channel congestion, whereas deliberately striping traffic across multiple nodes could recruit additional aggregate memory bandwidth. Developing runtime systems that dynamically monitor memory controller queue depths and adapt swizzle-striping granularity remains an open challenge.

7.3 Conclusion

The rapid growth of data-intensive workloads has propelled distributed computing systems to unprecedented scales. Yet petabyte-scale global memory cannot be managed the same way as a traditional single-node system because of its unprecedented scale: address translation is bounded by limited TLB reach with page translation, and a centralized global allocator cannot handle diverse memory allocation patterns across applications with high throughput.

This dissertation has demonstrated that hardware address translation and software memory management are two complementary halves to managing a global memory system:

- By pairing variable-sized segment descriptors with compact 64-bit swizzle masks, **UD-Trans** breaks the coupling between translation reach and data placement, delivering unbounded virtual memory reach with compact translation state, 1-cycle translation, and software-controlled locality control.

- Via a two-level control scheme that separates privileged physical memory isolation from unprivileged massively parallel allocations, **GLMalloc** eliminates centralized coordination bottlenecks, delivering atomic-free fast allocation with high-throughput and load-balancing under severe skewness.

Together, UDTrans and GLMalloc show that scale-out global memory systems can achieve performance and programmability simultaneously with a unified memory management design. By co-designing hardware translation and runtime memory management around a 2-dimensional physical address space, we can build extreme-scale systems that achieve not only the high throughput and energy efficiency of specialized hardware, but also the flexibility, safety, and programmability of unified virtual memory.

REFERENCES

[1] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. "Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 145–158. DOI: 10.1109/ISCA45697.2020.00023.

[2] Dennis A. Adams. *CRAY T3D System Architecture Overview*. Manual HR-04033. Available at https://www.bitsavers.org/pdf/cray/HR-04033_CRAY_T3D_System_Architecture_Overview_Sep93.pdf. Cray Research, Inc. Sept. 1993.

[3] Andrew V. Adinetz and Dirk Pleiter. *Halloc: A High-Throughput Dynamic Memory Allocator for GPGPU Architectures*. GPU Technology Conference (GTC) 2014, talk; source at https://github.com/canonizer/halloc. 2014.

[4] Tyler Allen, Bennett Cooper, and Rong Ge. "Fine-grain Quantitative Analysis of Demand Paging in Unified Virtual Memory". In: *ACM Trans. Archit. Code Optim.* 21.1 (Jan. 2024). ISSN: 1544-3566. DOI: 10.1145/3632953. URL: https://doi.org/10.1145/3632953.

[5] Tyler Allen and Rong Ge. "Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis". In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021, pp. 141–150. DOI: 10.1109/IPDPS49936.2021.00023.

[6] Tyler Allen and Rong Ge. "In-depth analyses of unified virtual memory system for GPU accelerated computing". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3480855. URL: https://doi.org/10.1145/3458817.3480855.

[7] George Almasi. "PGAS (Partitioned Global Address Space) Languages". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA, USA: Springer, 2011, pp. 1539–1545. DOI: 10.1007/978-0-387-09766-4_210.

[8] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. "The Tera Computer System". In: *Proc. 4th Int. Conf. Supercomputing (ICS)*. 1990, pp. 1–6. DOI: 10.1145/77726.255132.

[9] *Aurora Supercomputer*. https://www.alcf.anl.gov/aurora. 2024.

[10] John Bachan, Dan Bonachea, Paul H. Hargrove, Steve Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian van Straalen, and Scott B. Baden. "The UPC++ PGAS library for Exascale Computing". In: *Proceedings of the Second Annual PGAS Applications Workshop*. PAW17. Denver, CO, USA: Association for Computing Machinery, 2017. ISBN: 9781450351232. DOI: 10.1145/3144779.3169108. URL: https://doi.org/10.1145/3144779.3169108.

[11] Thomas W. Barr, Alan L. Cox, and Scott Rixner. "Translation caching: skip, don't walk (the page table)". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: Association for Computing Machinery, 2010, pp. 48–59. ISBN: 9781450300537. DOI: 10.1145/1815961.1815970. URL: https://doi.org/10.1145/1815961.1815970.

[12] Trinayan Baruah, Yifan Sun, Ali Tolga Dinçer, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. "Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 596–609. DOI: 10.1109/HPCA47549.2020.00055.

[13] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. "Efficient virtual memory for big memory servers". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: Association for Computing Machinery, 2013, pp. 237–248. ISBN: 9781450320795. DOI: 10.1145/2485922.2485943. URL: https://doi.org/10.1145/2485922.2485943.

[14] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: Expressing Locality and Independence with Logical Regions". In: *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*. Salt Lake City, UT, USA, 2012. DOI: 10.1109/SC.2012.71.

[15] Michael Beebe, Benjamin Michalowicz, Andrew McNamara, Yash Kumar, Dhabaleswar K. Panda, Yong Chen, Wendy Poole, and Steve Poole. "OpenSHMEM MLIR: A Dialect for Compile-Time Optimization of One-Sided Communications". In: *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC Workshops '25. Association for Computing Machinery, 2025, pp. 1075–1087. ISBN: 9798400718717. DOI: 10.1145/3731599.3767483. URL: https://doi.org/10.1145/3731599.3767483.

[16] A. Bensoussan, C. T. Clingen, and R. C. Daley. "The Multics virtual memory: concepts and design". In: *Commun. ACM* 15.5 (May 1972), pp. 308–318. ISSN: 0001-0782. DOI: 10.1145/355602.361306. URL: https://doi.org/10.1145/355602.361306.

[17] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. "Hoard: A Scalable Memory Allocator for Multithreaded Applications". In: *Proc. 9th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. Cambridge, MA, USA, Nov. 2000. DOI: 10.1145/378993.379232.

[18] Abhishek Bhattacharjee. "Large-reach memory management unit caches". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: Association for Computing Machinery, 2013, pp. 383–394. ISBN: 9781450326384. DOI: 10.1145/2540708.2540741. URL: https://doi.org/10.1145/2540708.2540741.

[19] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan RK Ports. "PRISM: Rethinking the RDMA interface for distributed systems". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 228–242.

[20] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. "Efficient Distributed Memory Management with RDMA and Caching". In: *Proc. VLDB Endowment* 11.11 (2018).

[21] B.L. Chamberlain, D. Callahan, and H.P. Zima. "Parallel Programmability and the Chapel Language". In: *Int. J. High Perform. Comput. Appl.* 21.3 (Aug. 2007), pp. 291–312. ISSN: 1094-3420. DOI: 10.1177/1094342007078442. URL: https://doi.org/10.1177/1094342007078442.

[22] Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi. "User-Defined Distributions and Layouts in Chapel: Philosophy and Framework". In: *Proc. 2nd USENIX Workshop on Hot Topics in Parallelism (HotPAR)*. Berkeley, CA, USA, June 2010.

[23] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing". In: *Proc. 20th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. San Diego, CA, USA, 2005. DOI: 10.1145/1094811.1094852.

[24] Dongwei Chen, Dong Tong, Chun Yang, Jiangfang Yi, and Xu Cheng. "FlexPointer: Fast Address Translation Based on Range TLB and Tagged Pointers". In: *ACM Trans. Archit. Code Optim.* 20.2 (Mar. 2023). ISSN: 1544-3566. DOI: 10.1145/3579854. URL: https://doi.org/10.1145/3579854.

[25] Andrew A. Chien, Charles Colley, Jianru Ding, Alexander Fell, David F. Gleich, Henry Hoffmann, Moubarak Jeje, Rajat Khandelwal, Yanjing Li, Jose M. Monsalve-Diaz, Marziyeh Nourian, Andronicus Rajasukumar, Jiya Su, Tianshuo Su, Yuqing Wang, Wenyi Wang, Ruiqi Xu, and Tianchi Zhang. "UpDown: A Supercomputer Co-Designed for Scalable Graph Processing". In: *IEEE Transactions on Parallel and Distributed Systems* 37 (2026), pp. 1553–1570. DOI: 10.1109/TPDS.2026.3681212. URL: https://api.semanticscholar.org/CorpusID:281712032.

[26] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. "ScaLAPACK: a portable linear algebra library for distributed memory computers — design issues and performance". In: *Computer Physics Communications* 97.1 (1996). High-Performance Computing in Science, pp. 1–15.

ISSN: 0010-4655. DOI: 10.1016/0010-4655(96)00017-3. URL: https://www.sciencedirect.com/science/article/pii/0010465596000173.

[27] The Linux kernel development community. *HugeTLB pages*. 2024. URL: https://kernel.org/doc/html/v6.9-rc7/admin-guide/mm/hugetlbpage.html.

[28] The Linux kernel development community. *Transparent Hugepage Support*. 2024. URL: https://www.kernel.org/doc/html/v6.9-rc7/admin-guide/mm/transhuge.html.

[29] Bennett Cooper, Thomas RW Scogland, and Rong Ge. "Shared Virtual Memory: Its Design and Performance Implications for Diverse Applications". In: *Proceedings of the 38th ACM International Conference on Supercomputing*. ICS '24. Kyoto, Japan: Association for Computing Machinery, 2024, pp. 26–37. ISBN: 9798400706103. DOI: 10.1145/3650200.3656608. URL: https://doi.org/10.1145/3650200.3656608.

[30] AMD Corporation. *AMD64 Architecture Programmer's Manual*. 2024. URL: https://www.amd.com/en/search/documentation/hub.html.

[31] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer Manuals*. 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[32] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. "FaRM: Fast Remote Memory". In: *Proc. 11th USENIX Symp. Networked Systems Design and Implementation (NSDI)*. Seattle, WA, USA, 2014. URL: https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-dragojevic.pdf.

[33] P. Erdős and A. Rényi. "On the evolution of random graphs". In: *The Structure and Dynamics of Networks*. Princeton: Princeton University Press, 2006, pp. 38–82. ISBN: 9781400841356. DOI: 10.1515/9781400841356.38. URL: https://doi.org/10.1515/9781400841356.38.

[34] Jason Evans. "A Scalable Concurrent malloc(3) Implementation for FreeBSD". In: *Proceedings of the 2006 BSDCan Conference*. Ottawa, Canada, May 2006. URL: https://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf.

[35] Mohammad Ewais and Paul Chow. "Disaggregated Memory in the Datacenter: A Survey". In: *IEEE Access* 11 (2023), pp. 20688–20712. DOI: 10.1109/ACCESS.2023.3250407.

[36] John Feo, David Harper, Simon Kahan, and Petr Konecny. "ELDORADO". In: *Proc. 2nd Conf. Computing Frontiers (CF)*. 2005, pp. 28–34. DOI: 10.1145/1062261.1062268.

[37] Tais B. Ferreira, Rivalino Matias, Autran Macêdo, and Lucio B. Araujo. "A Comparison of Memory Allocators for Multicore and Multithread Applications: A Quantitative Approach". In: *2011 Brazilian Symposium on Computing System Engineering*. 2011, pp. 200–205. DOI: 10.1109/SBESC.2011.29.

[38] Philip Werner Frey and Gustavo Alonso. "Minimizing the hidden cost of RDMA". In: *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE. 2009, pp. 553–560.

[39] Luigi Fusco, Mikhail Khalilov, Marcin Chrapek, Giridhar Chukkapalli, Thomas Schulthess, and Torsten Hoefler. *Understanding Data Movement in Tightly Coupled Heterogeneous Systems: A Case Study with the Grace Hopper Superchip*. 2024. arXiv: 2408.11556 [cs.DC]. URL: https://arxiv.org/abs/2408.11556.

[40] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. "Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory". In: *Proc. ACM/IEEE 46th Annual Int. Symp. Computer Architecture (ISCA)*. Phoenix, AZ, USA, 2019.

[41] Bin Gao, Qingxuan Kang, Hao-Wei Tee, Kyle Timothy Ng Chu, Alireza Sanaee, and Djordje Jevdjic. "Scalable and Effective Page-table and TLB management on NUMA Systems". In: *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 445–461. ISBN: 978-1-939133-41-0. URL: https://www.usenix.org/conference/atc24/presentation/gao-bin-scalable.

[42] Tarek El-Ghazawi and François Cantonnet. "UPC Performance and Potential: A NPB Experimental Study". In: *Proc. 2002 ACM/IEEE Conf. Supercomputing (SC '02)*. Baltimore, MD, 2002.

[43] Wolfram Gloger. *ptmalloc3: A Multi-Threaded Memory Allocator*. http://www.malloc.de/en/. 2006.

[44] *Introducing Trillium, 6th Generation TPUs*. https://cloud.google.com/blog/products/compute/introducing-trillium-6th-gen-tpus. 2024.

[45] Google Cloud. *AI Hypercomputer Overview*. Google Cloud documentation, https://docs.cloud.google.com/ai-hypercomputer/docs/overview. Accessed 2025-09-14. 2025.

[46] Mel Gorman. *Foundation for Automatic NUMA Balancing*. Linux kernel mailing list, reposted at https://lwn.net/Articles/523065/. Online. Nov. 2012.

[47] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. "Direct Access, High-Performance Memory Disaggregation with DirectCXL". In: *Proc. USENIX Annual Technical Conference (ATC)*. Carlsbad, CA, USA, 2022, pp. 287–294.

[48] Graph500 Steering Committee. *Graph500 Benchmark*. https://graph500.org/. Accessed: 2024-09-18.

[49] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. "GraphBIG: understanding graph computing in the context of industrial solutions". In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12. DOI: 10.1145/2807591.2807626.

[50] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. "Rebooting Virtual Memory with Midgard". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 512–525. DOI: 10.1109/ISCA52012.2021.00047.

[51] Congjie He, Yeqi Huang, Pei Mu, Ziming Miao, Jilong Xue, Lingxiao Ma, Fan Yang, and Luo Mai. "WaferLLM: large language model inference at wafer scale". In: *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*. OSDI '25. Boston, MA, USA: USENIX Association, 2025. ISBN: 978-1-939133-47-2.

[52] Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen-mei Hwu. "XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines". In: *Proc. 10th IEEE Int. Conf. Computer and Information Technology (CIT)*. 2010, pp. 1134–1139. DOI: 10.1109/CIT.2010.206.

[53] Intel. *Intel Gaudi 3 AI Accelerator*. 2024. URL: https://www.intel.com/content/www/us/en/products/details/processors/ai-accelerators/gaudi3.html.

[54] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. "DUCATI: High-performance Address Translation by Extending TLB Reach of GPU-accelerated Systems". In: *ACM Trans. Archit. Code Optim.* 16.1 (Mar. 2019). ISSN: 1544-3566. DOI: 10.1145/3309710. URL: https://doi.org/10.1145/3309710.

[55] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. "A domain-specific supercomputer for training deep neural networks". In: *Commun. ACM* 63.7 (June 2020), pp. 67–78. ISSN: 0001-0782. DOI: 10.1145/3360307. URL: https://doi.org/10.1145/3360307.

[56] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 1–12. ISSN: 0163-5964. DOI: 10.1145/3140659.3080246. URL: https://doi.org/10.1145/3140659.3080246.

[57] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. "Redundant memory mappings for fast access to large memories". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. Portland, Oregon: Association for Computing Machinery, 2015, pp. 66–78. ISBN: 9781450334020. DOI: 10.1145/2749469.2749471. URL: https://doi.org/10.1145/2749469.2749471.

[58] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. "Big data causing big (TLB) problems: taming random memory accesses on the GPU". In: *Proceedings of the 13th International Workshop on Data Management on New Hardware*. DAMON '17. Chicago, Illinois: Association for Computing Machinery, 2017. ISBN: 9781450350259. DOI: 10.1145/3076113.3076115. URL: https://doi.org/10.1145/3076113.3076115.

[59] R. E. Kessler and J. L. Schwarzmeier. "CRAY T3D: A New Dimension for Cray Research". In: *Digest of Papers, COMPCON Spring '93*. 1993, pp. 176–182. DOI: 10.1109/CMPCON.1993.289660.

[60] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. "Batch-Aware Unified Memory Management in GPUs for Irregular Workloads". In: *Proc. 25th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland, 2020. DOI: 10.1145/3373376.3378529.

[61] Andi Kleen. *A NUMA API for Linux*. Whitepaper. https://halobates.de/numaapi3.pdf. SUSE Labs, Aug. 2004.

[62] Marcin Knap and Paweł Czarnul. "Performance Evaluation of Unified Memory with Prefetching and Oversubscription for Selected Parallel CUDA Applications on NVIDIA Pascal and Volta GPUs". In: *J. Supercomputing* 75.11 (2019), pp. 7625–7645. DOI: 10.1007/s11227-019-02966-8.

[63] Nikhita Kunati and Michael M Swift. "Implementation of Direct Segments on a RISC-V Processor". en. In: *Second Workshop on Computer Architecture Research with RISC-V (CARRV)*. Los Angeles, CA, USA, 2018.

[64] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 611–626. ISBN: 9798400702297. DOI: 10.1145/3600006.3613165. URL: https://doi.org/10.1145/3600006.3613165.

[65] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. "Coordinated and efficient huge page management with ingens". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 705–721. ISBN: 9781931971331.

[66] Kartik Lakhotia, Laura Monroe, Kelly Isham, Maciej Besta, Nils Blach, Torsten Hoefler, and Fabrizio Petrini. "PolarStar: Expanding the Horizon of Diameter-3 Networks". In: *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '24. Nantes, France: Association for Computing Machinery, 2024, pp. 345–357. ISBN: 9798400704161. DOI: 10.1145/3626183.3659975. URL: https://doi.org/10.1145/3626183.3659975.

[67] Christoph Lameter. "An Overview of Non-Uniform Memory Access". In: *Commun. ACM* 56.9 (Sept. 2013), pp. 59–65.

[68] J. Laudon and D. Lenoski. "System overview of the SGI Origin 200/2000 product line". In: *Proceedings IEEE COMPCON 97. Digest of Papers*. 1997, pp. 150–156. DOI: 10.1109/CMPCON.1997.584688.

[69] Dongjae Lee, Bongjoon Hyun, Youngjin Kwon, and Minsoo Rhu. "PIM-malloc: A Fast and Scalable Dynamic Memory Allocator for Processing-in-Memory (PIM) Architectures". In: *Proc. IEEE 32nd Int. Symp. High-Performance Computer Architecture (HPCA)*. Sydney, Australia, 2026. DOI: 10.1109/HPCA68181.2026.11408588. arXiv: 2505.13002.

[70] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. "Mimalloc: Free List Sharding in Action". In: *Proc. 17th Asian Symp. Programming Languages and Systems (APLAS)*. Vol. 11893. LNCS. Springer, 2019. DOI: 10.1007/978-3-030-34175-6_13.

[71] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. "Thread and Memory Placement on NUMA Systems: Asymmetry Matters". In: *Proc. 2015 USENIX Annu. Tech. Conf. (USENIX ATC '15)*. Santa Clara, CA, 2015, pp. 277–289.

[72] Bingyao Li, Yanan Guo, Yueqi Wang, Aamer Jaleel, Jun Yang, and Xulong Tang. "IDYLL: Enhancing Page Translation in Multi-GPUs via Light Weight PTE Invalidations". In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '23. Toronto, ON, Canada: Association for Computing Machinery, 2023, pp. 1163–1177. ISBN: 9798400703294. DOI: 10.1145/3613424.3614269. URL: https://doi.org/10.1145/3613424.3614269.

[73] Bingyao Li, Jieming Yin, Youtao Zhang, and Xulong Tang. "Improving Address Translation in Multi-GPUs via Sharing and Spilling aware TLB Design". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 1154–1168. ISBN: 9781450385572. DOI: 10.1145/3466752.3480083. URL: https://doi.org/10.1145/3466752.3480083.

[74] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. *Pond: CXL-Based Memory Pooling Systems for Cloud Platforms*. 2022. arXiv: 2203.00241 [cs.OS]. URL: https://arxiv.org/abs/2203.00241.

[75] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. "CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving". In: *Proceedings of the ACM SIGCOMM 2024 Conference*. ACM SIGCOMM '24. Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 38–56. ISBN: 9798400706141. DOI: 10.1145/3651890.3672274. URL: https://doi.org/10.1145/3651890.3672274.

[76] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. "Prefetched Address Translation". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 1023–1036. ISBN: 9781450369381. DOI: 10.1145/3352460.3358294. URL: https://doi.org/10.1145/3352460.3358294.

[77] Hasan Al Maruf and Mosharaf Chowdhury. "Memory Disaggregation: Advances and Open Challenges". In: *arXiv preprint* (2023). arXiv: 2305.03943.

[78] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. "TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory". In: *Proc. 28th ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Vol. 3. Vancouver, Canada, 2023. DOI: 10.1145/3582016.3582063.

[79] Hunter McCoy and Prashant Pandey. "Gallatin: A General-Purpose GPU Memory Manager". In: *Proc. 29th ACM SIGPLAN Annual Symp. Principles and Practice of Parallel Programming (PPoPP)*. 2024, pp. 364–376. DOI: 10.1145/3627535.3638499.

[80] Salman Mohamadi, Ghulam Mujtaba, Ngan Le, Gianfranco Doretto, and Donald Adjeroh. *ChatGPT in the Age of Generative AI and Large Language Models: A Concise Survey*. July 2023. DOI: 10.48550/arXiv.2307.04251.

[81] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. *CACTI 6.0: A Tool to Model Large Caches*. Tech. rep. HPL-2009-85. HP Laboratories, 2009.

[82] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. "Latency-Tolerant Software Distributed Shared Memory". In: *Proc. USENIX Annual Technical Conference (ATC)*. Santa Clara, CA, USA, 2015. URL: https://www.usenix.org/system/files/conference/atc15/atc15-paper-nelson_update.pdf.

[83] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers". In: *J. Supercomput.* 10.2 (1996), pp. 169–189.

[84] Robert W. Numrich and John Reid. "Co-Array Fortran for Parallel Programming". In: *ACM SIGPLAN Fortran Forum* 17.2 (1998), pp. 1–31. DOI: 10.1145/289918.289920.

[85] *NVlink and NVswitch (5th Generation)*. https://www.nvidia.com/en-us/data-center/nvlink/. 2024.

[86] Nvidia. *Nvidia DGX SuperPod with Grace Blackwell (GB200) chips*. 2024. URL: https://www.nvidia.com/en-us/data-center/dgx-superpod-gb200/.

[87] Nvidia. *NVIDIA Grace Hopper Superchip Architecture Whitepaper*. 2024. URL: https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper.

[88] Nvidia. *NVIDIA H100 Tensor Core GPU Architecture Overview Whitepaper*. 2024. URL: https://resources.nvidia.com/en-us-tensor-core.

[89] Nvidia. *The NVLink Network Switch*. 2022. URL: https://hc34.hotchips.org/assets/program/conference/day2/Network%20and%20Switches/NVSwitch%20HotChips%202022%20r5.pdf.

[90] Geraldo F. Oliveira, Emanuele G. Esposito, Juan Gómez-Luna, and Onur Mutlu. "PUMA: Efficient and Low-Cost Memory Allocation and Alignment Support for Processing-Using-Memory Architectures". In: *arXiv preprint* (Mar. 2024). arXiv: 2403.04539.

[91] *Frontier Supercomputer*. https://www.olcf.ornl.gov/olcf-resources/compute-systems/frontier/. 2022.

[92] Susan Ostrouchov. *Block Cyclic Data Distribution*. Netlib, https://www.netlib.org/utk/papers/factor/node3.html. Accessed 2024-08-10. 1995.

[93] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. "Every walk's a hit: making page walks single-access cache hits". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 128–141. ISBN: 9781450392051. DOI: 10.1145/3503222.3507718. URL: https://doi.org/10.1145/3503222.3507718.

[94] Gregory F. Pfister, William C. Brantley, David A. George, Steve L. Harvey, Walter J. Kleinfelder, Kevin P. McAuliffe, Evelin S. Melton, V. Alan Norton, and Jodi Weiss. "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture". In: *Proc. Int. Conf. Parallel Processing (ICPP)*. Aug. 1985, pp. 764–771.

[95] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. "CoLT: Coalesced Large-Reach TLBs". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 2012, pp. 258–269. DOI: 10.1109/MICRO.2012.32.

[96] Jason Power, Mark D. Hill, and David A. Wood. "Supporting x86-64 address translation for 100s of GPU lanes". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014, pp. 568–578. DOI: 10.1109/HPCA.2014.6835965.

[97] Raghu Prabhakar, Ram Sivaramakrishnan, Darshan Gandhi, Yun Du, Mingran Wang, Xiangyu Song, Kejie Zhang, Tianren Gao, Angela Wang, Karen Li, Yongning Sheng, Joshua Brot, Denis Sokolov, Apurv Vivek, Calvin Leung, Arjun Sabnis, Jiayu Bai, Tuowen Zhao, Mark Gottscho, David Jackson, Mark Luttrell, Manish K. Shah, Edison Chen, Kaizhao Liang, Swayambhoo Jain, Urmish Thakker, Dawei Huang, Sumti Jairath, Kevin J. Brown, and Kunle Olukotun. *SambaNova SN40L: Scaling the AI Memory Wall with Dataflow and Composition of Experts*. 2024. arXiv: 2405.07518 [cs.AR]. URL: https://arxiv.org/abs/2405.07518.

[98] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. "vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Rotterdam, Netherlands, 2025. DOI: 10.1145/3669940.3707256.

[99] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. "GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 908–921. DOI: 10.1109/MICRO50266.2020.00078.

[100] Andronicus Rajasukumar, Jiya Su, Yuqing Wang, Tianshuo Su, Marziyeh Nourian, Jose M Monsalve Diaz, Tianchi Zhang, Jianru Ding, Wenyi Wang, Ziyi Zhang, Moubarak Jeje, Henry Hoffmann, Yanjing Li, and Andrew A. Chien. *UpDown: Programmable fine-grained Events for Scalable Performance on Irregular Applications*. 2024. arXiv: 2407.20773 [cs.AR]. URL: https://arxiv.org/abs/2407.20773.

[101] Andronicus Rajasukumar, Ruiqi Xu, Tianchi Zhang, Yuqing (Ivy) Wang, Tianshuo Su, Marziyeh Nourian, Jianru Ding, Jiya Su, Rajat Khandelwal, Alexander Fell, David F. Gleich, Yanjing Li, Henry Hoffmann, and Andrew A. Chien. "UpDown: Efficient Manycore based on Many Threading and Scalable Memory Parallelism". In: *Proceedings of the 40th ACM International Conference on Supercomputing*. ICS '26. Association for Computing Machinery, 2026, pp. 603–617. ISBN: 9798400725227. DOI: 10.1145/3797905.3800534. URL: https://doi.org/10.1145/3797905.3800534.

[102] Andronicus Rajasukumar, Tianchi Zhang, Ruiqi Xu, and Andrew A. Chien. "UpDown: A Novel Architecture for Unlimited Memory Parallelism". In: *Proceedings of the International Symposium on Memory Systems*. Washington, DC, USA: ACM, 2024, pp. 61–77. DOI: 10.1145/3695794.3695801.

[103] *Fugaku Supercomputer*. https://www.r-ccs.riken.jp/en/fugaku/. 2020.

[104] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. "Recency-based TLB preloading". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ISCA '00. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2000, pp. 117–127. ISBN: 1581132328. DOI: 10.1145/339647.339666. URL: https://doi.org/10.1145/339647.339666.

[105] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. "Scalable Locality-Conscious Multithreaded Memory Allocation". In: *Proc. 5th Int. Symp. Memory Management (ISMM)*. Ottawa, Canada, 2006, pp. 84–94. DOI: 10.1145/1133956.1133968.

[106] Steven L. Scott. "Synchronization and Communication in the T3E Multiprocessor". In: *Proc. 7th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. 1996, pp. 26–36. DOI: 10.1145/237090.237144.

[107] Steven L. Scott and Gregory M. Thorson. *Recursive Address Centrifuge for Distributed Memory Massively Parallel Processing Systems*. U.S. Patent 5,696,922. Assignee: Cray Research, Inc. Filed Dec. 10, 1993; issued Dec. 9, 1997. Dec. 1997.

[108] Matthias Springer and Hidehiko Masuhara. "DynaSOAr: A Parallel Memory Allocator for Object-Oriented Programming on GPUs with Efficient Memory Access". In: *33rd European Conf. Object-Oriented Programming (ECOOP 2019)*. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). 2019, 17:1–17:37. DOI: 10.4230/LIPIcs.ECOOP.2019.17.

[109] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. "ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU". In: *Innovative Parallel Computing (InPar)*. 2012, pp. 1–10. DOI: 10.1109/InPar.2012.6339604.

[110] Michael M. Swift. "Towards O(1) Memory". In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. Whistler, BC, Canada: Association for Computing Machinery, 2017, pp. 7–11. ISBN: 9781450350686. DOI: 10.1145/3102980.3102982. URL: https://doi.org/10.1145/3102980.3102982.

[111] Raymond D. Turney. *Comparison of 250 MHz R10K Origin 2000 and 400 MHz R12K Origin 2000 Using NAS Parallel Benchmarks*. Tech. rep. NAS-01-007. Moffett Field, CA, USA: NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center, 2001. URL: https://www.nas.nasa.gov/assets/nas/pdf/techreports/2001/nas-01-007.pdf.

[112] Yongbin Gu, Wenxuan Wu, Yunfan Li, and Lizhong Chen. "UVMBench: A Comprehensive Benchmark Suite for Researching Unified Virtual Memory in GPUs". In: *CoRR* abs/2007.09822 (2020). arXiv: 2007.09822. URL: https://arxiv.org/abs/2007.09822.

[113] David W Walker and Jack J Dongarra. "MPI: a standard message passing interface". In: *Supercomputer* 12 (1996), pp. 56–68.

[114] Luming Wang, Xu Zhang, Songyue Wang, Zhuolun Jiang, Tianyue Lu, Mingyu Chen, Siwei Luo, and Keji Huang. "Asynchronous Memory Access Unit: Exploiting Massive Parallelism for Far Memory Access". In: *ACM Trans. Archit. Code Optim.* 21.3 (Sept. 2024). ISSN: 1544-3566. DOI: 10.1145/3663479. URL: https://doi.org/10.1145/3663479.

[115] Xiaoyuan Wang, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. "TLB Coalescing for Multi-Grained Page Migration in Hybrid Memory Systems". In: *IEEE Access* 8 (2020). Conference Name: IEEE Access, pp. 66304–66314. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2983065. URL: https://ieeexplore.ieee.org/document/9046005/?arnumber=9046005 (visited on 07/15/2024).

[116] Yuqing Wang, Swann Perarnau, and Andrew A. Chien. "UpDown: Combining Scalable Address Translation with Locality Control". In: *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Atlanta, GA, USA: IEEE, Nov. 2024, pp. 1014–1024. DOI: 10.1109/SCW63240.2024.00141.

[117] Martin Winter, Daniel Mlakar, Mathias Parger, and Markus Steinberger. "Ouroboros: Virtualized Queues for Dynamic Memory Management on GPUs". In: *Proc. 34th ACM Int. Conf. Supercomputing (ICS)*. 2020, pp. 1–12. DOI: 10.1145/3392717.3392742.

[118] Martin Winter, Mathias Parger, Daniel Mlakar, and Markus Steinberger. "Are Dynamic Memory Managers on GPUs Slow? A Survey and Benchmarks". In: *Proc. 26th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*. 2021, pp. 219–233. DOI: 10.1145/3437801.3441612.

[119] Tim S Woodall, Galen M Shipman, George Bosilca, Richard L Graham, and Arthur B Maccabe. "High performance RDMA protocols in HPC". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 13th European PVM/MPI User's Group Meeting Bonn, Germany, September 17-20, 2006 Proceedings 13*. Springer. 2006, pp. 76–85.

[120] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis". In: *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. Kyoto, 2014. URL: https://www.mcs.anl.gov/papers/P5064-0114.pdf.

[121] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. "Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 339–351. DOI: 10.1109/MICRO.2018.00035.

[122] Chao Yu, Yuebin Bai, and Rui Wang. *Enabling Large-Reach TLBs for High-Throughput Processors by Exploiting Memory Subregion Contiguity*. 2021. arXiv: 2110.08613 [cs.AR]. URL: https://arxiv.org/abs/2110.08613.

[123] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos. "Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA '23. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: 10.1145/3579371.3589079. URL: https://doi.org/10.1145/3579371.3589079.

[124] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. "Towards High Performance Paged Memory for GPUs". In: *Proc. IEEE Int. Symp. High Performance Computer Architecture (HPCA)*. Barcelona, Spain, 2016. DOI: 10.1109/HPCA.2016.7446077.

[125] Zhuangzhuang Zhou, Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, Christina Delimitrou, and Parthasarathy Ranganathan. "Characterizing a Memory Allocator at Warehouse Scale". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 192–206. ISBN: 9798400703867. DOI: 10.1145/3620666.3651350. URL: https://doi.org/10.1145/3620666.3651350.