PIMCloud: QoS-Aware Resource Management of Latency-Critical Applications in Clouds with Processing-in-Memory

Shuang Chen, Yi Jiang, Christina Delimitrou, José F. Martínez
Computer Systems Laboratory
Cornell University
Ithaca, NY, USA
{sc2682,yj389,delimitrou,martinez}@cornell.edu

***Abstract*— The slowdown of Moore's Law, combined with advances in 3D stacking of logic and memory, have pushed architects to revisit the concept of processing-in-memory (PIM) to overcome the memory wall bottleneck. This PIM renaissance finds itself in a very different computing landscape from the one twenty years ago, as more and more computation shifts to the cloud. Most PIM architecture papers still focus on best-effort applications, while PIM's impact on latency-critical cloud applications is not well understood.**

This paper explores how datacenters can exploit PIM architectures in the context of latency-critical applications. We adopt a general-purpose cloud server with HBM-based, 3D-stacked logic+memory modules, and study the impact of PIM on six diverse interactive cloud applications. We reveal the previously neglected opportunity that PIM presents to these services, and show the importance of properly managing PIM-related resources to meet the QoS targets of interactive services and maximize resource efficiency. Then, we present PIMCloud, a QoS-aware resource manager designed for cloud systems with PIM allowing colocation of multiple latency-critical and best-effort applications. We show that PIMCloud efficiently manages PIM resources: it (1) improves effective machine utilization by up to 70% and 85% (average 24% and 33%) under 2-app and 3-app mixes, compared to the best state-of-the-art manager; (2) helps latency-critical applications meet QoS; and (3) adapts to varying load patterns.

I. Introduction

The slowdown of Moore's Law demands novel architectures to push performance, such as 3D stacking [4, 5] which has revitalized the concept of processing-in-memory (PIM) as a means to tackle the von Neumann bottleneck, resulting in a wide range of recent PIM architecture proposals [9, 11, 12, 13, 17, 23, 33, 35, 37, 38, 45, 47, 53, 61], along with a major DARPA/SRC research center for building a PIM ecosystem [66].

At the same time, cloud computing is becoming ubiquitous, offering resource flexibility and cost efficiency [15]. An increasing amount of computing now takes place in large-scale datacenters [15, 42, 62]. These two trends point to a near future where datacenter nodes will incorporate PIM capabilities, which makes it imperative to examine PIM's role in cloud servers. However, in order for datacenter nodes to leverage PIM in a way that benefits application performance, there are a number of challenges that remain unsolved.

First, the diversity of cloud applications is rapidly increasing [42], making it impractical to have one specialized PIM architecture tailored to each workload type [9, 13, 23, 37]. For a PIM architecture to be practical enough to be deployed at scale, it should be able to handle general-purpose computation of cloud services. Therefore, in line with recent proposals for general-purpose PIM architectures [17, 34, 46, 49, 70], we target datacenter nodes embedded with multiple low-latency 3D memory+logic stacks, a number of low-power general-purpose cores embedded in the logic layer of each stack, and a memory abstraction shared with the main CPU.

Second, current PIM research typically targets memory-intensive workloads, where the high memory bandwidth of PIM helps improve performance for throughput-bound, best-effort (BE) jobs [35, 45, 51, 52, 53, 61]. However, datacenters host a different type of applications where latency, not throughput, is the primary performance metric [22, 41, 44]. These latency-critical (LC) services, like websearch and key-value stores, are not necessarily memory bandwidth-intensive, but they require low memory latency. Latency is also becoming more important as a metric due to the increasing prevalence of microservices [32, 50, 68], which impose strict, often microsecond-level, quality-of-service (QoS) constraints in terms of tail latency [27]. Thus, for PIM to gain wide adoption in the cloud, it is important to quantify its impact on such latency-critical applications.

In this paper, we perform a comprehensive study of PIM on LC applications, and quantify the impact of PIM's reduced memory latency, shallow memory hierarchy, and simple core architecture. We observe that many LC services favor PIM, compared to an iso-silicon architecture with brawny cores, a deep memory hierarchy, and higher memory latency.

Finally, given the opportunity of PIM to LC applications, it is also important to design resource management techniques that are aware of a system's PIM capabilities, and can allocate the right resources to each LC application. This is especially critical under colocation, where cloud operators often co-schedule multiple LC/BE applications on the same physical

server, to improve resource and cost efficiency [19, 20, 21, 51, 52, 54, 55, 59, 69, 74]. Without proper resource management to eliminate contention, LC applications can experience QoS violations.

To address these challenges, we design PIMCloud, a PIM-aware and QoS-aware resource manager for LC applications in PIM-enabled systems. PIMCloud leverages the varying degree of benefits that PIM brings to different LC applications, and assigns the most suitable resources to each colocated application. To this end, it manages PIM-introduced resources including heterogeneous cores and data placement. PIMCloud additionally adjusts resource allocations dynamically, to cope with the varying load patterns of LC applications.

PIMCloud identifies the opportunity that PIM presents for latency-critical applications, and manages PIM resources to maximize their benefit for this emerging type of cloud services. We evaluate PIMCloud using a cycle-level multicore simulator [63] under various scenarios. Evaluation results show that PIMCloud effectively manages PIM-related resources while meeting QoS, and improves effective machine utilization by up to 70% and 85% (average 24% and 33%) under 2-app and 3-app mixes, compared to the best state-of-the-art manager.

II. Related Work

Prior PIM proposals target throughput-oriented applications, including data analytics [34], graph processing [10], MapReduce[61], bulk bitwise operations in databases [64], and machine learning [17, 23, 46, 49]. PIMCloud instead explores, for the first time, the role of PIM for LC applications.

A. Resource Management in PIM-enabled Systems

Integrating PIM in conventional CPU-based systems complicates resource management. Liu et al. [49] adopt a system for neural network (NN) training with conventional CPUs, general-purpose, and specialized PIM cores. They profile the memory:compute ratio of each NN operator, and schedule them to different resources to improve system utilization. Tsai et al. propose AMS [70], a dynamic thread scheduler for systems with a host CPU and general-purpose PIM stacks. The scheduler leverages cache monitoring to estimate the utility of PIM over CPU, and schedules threads one at a time to their preferred resource to improve overall weighted speedup [67]. All these systems are designed to improve overall system throughput. We show in this paper that LC applications present different challenges for PIM when the optimization goal is not simply throughput, and a redesign of the resource manager is required to meet LC applications' strict latency constraints.

B. Resource Management for LC Applications

There has been abundant work on resource management of LC applications in conventional multicores. Most restricts a single LC application per node, while more recent work allows colocation of multiple LC applications [21, 55, 59]. Compared to conventional architectures, PIM introduces both core and memory heterogeneity which requires careful management.

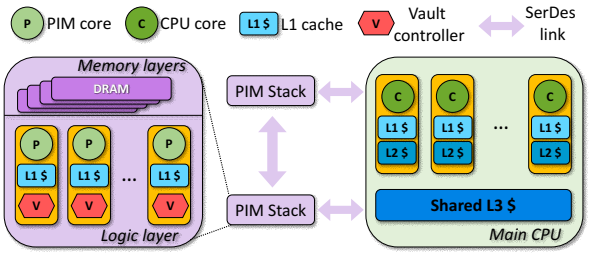


Fig. 1: Envisioned PIM-enabled cloud server.

1) Core Management in Heterogenous Systems: Prior work has extensively studied core management for heterogeneous systems [25, 54, 58, 60, 71, 73], such as big.LITTLE, but mostly for throughput-oriented applications. Octopus-Man [60] is a state-of-the-art core manager for LC applications, and is most related to our work. However, because it optimizes for energy consumption, the core selection algorithm is rather simple: it always selects small cores first, then big. It also restricts the number of LC applications to only one per node. There has been no related work that targets performance-oriented colocation of LC applications in a heterogenous environment. While PIMCloud focuses on PIM, its techniques can also be applied to systems like big.LITTLE.

2) Data Placement in NUMA Systems: Data placement in a NUMA system determines where the OS places memory pages [18, 26, 56, 72]. As we show in Section III-A, when memory stacks are augmented with computation capabilities, data access by PIM cores within and across stacks may resemble a (clustered) NUMA architecture. Data placement for LC interactive services has only been studied in a small number of previous proposals. For example, RackOut [57] and Scale-Out ccNUMA [36] propose data aggregation techniques and software caching of hot items for key-value stores to improve data locality. Prior work has not, however, studied data placement under colocation or varying resource allocations for LC applications. The distinguishing feature that LC services bring to the system is that because LC applications have fluctuating loads, core allocation changes dynamically at runtime, which may require memory pages to also move frequently. It is critical for data placement to be aware of the real-time core allocation to reduce the amount of dynamic page movement. This becomes even more critical in PIM-enabled systems, because memory capacity per PIM stack (a few GBs) is much less than a NUMA node (a few hundred of GBs), leading to more memory contention and data movement.

III. Implications of PIM to LC Applications

In this section, we first provide an overview of the PIM-enabled system architecture we target, a general-purpose datacenter node with PIM capabilities. Then, we study the impact of a variety of factors introduced by PIM to six diverse LC applications on both latency and throughput.

TABLE I: System Specification

Brawn/CPU Core	Haswell-like [70], 2.4 GHz, 4-way issue, 60-entry IQ, 192-entry ROB, 72-entry LQ, 42-entry SQ, 2-level branch predictor with 1,024 18-bit BHSRs and 4,096 2-bit PHT entries, $13mm^2$ per core (plus its private caches) [3].
L1 cache	32/32 KB private instruction/data cache, 8-way set-associative, 3-cycle latency,
L2 cache	256 KB private, 8-way set-associative, 7-cycle latency.
L3 cache	2MB per slice (i.e. per CPU core) shared, 8-way set-associative, 30-cycle latency, $7mm^2$ per slice [3].
Memory	8 GB per memory stack, HBM-like organization [4], t_{CK}=0.8 ns, t_{RCD}=13.75 ns, t_{WR}=15 ns,t_{CL}=13.75 ns, t_{RAS}=27.5 ns, t_{RP}=13.75 ns, t_{REFI}=1.95 ns [39], $50mm^2$ area budget for components other than vault controllers and interconnect [70].
Wimpy/PIM Core	ARM Cortex A57-like [1], 2 GHz, each with a 32/32 KB private i-/d-cache, $5mm^2$ per core (plus its private caches) [2].
SerDes link	160 GBps bidirectional, 4 ns latency [70]

A. System Architecture Overview

Fig. 1 is a simplified diagram of the PIM-enabled datacenter server that we target. On the right is the main CPU, containing a traditional multicore processor with two levels of private caches and a shared last-level cache (LLC). The CPU connects to multiple memory devices using high-speed, high-bandwidth SerDes links. Conventional DDR modules are replaced by 3D memory stacks, whose logic layers are endowed with several simple general-purpose cores (PIM cores). Such PIM stacks constitute the CPU's main memory, but their embedded cores can also run programs independently. Stacks are connected to each other via SerDes links, such that PIM cores can access memory pages in other PIM stacks.

Because of power, area, and technology constraints in the memory stacks, PIM cores are relatively small and low-power, akin to Intel's Silvermont [70] or ARM's Cortex [49]. Meanwhile, CPU cores are larger and higher-performance, along the lines of Intel's Xeon [6] or Cavium's ThunderX2 [8]. We assume that all processing elements are ISA-compatible, similar to prior work [17, 34, 49, 70]. We also assume that all cores are governed by a single OS, sharing a single address space with identical page tables [14]. Unlike an accelerator setup where a "master" thread runs on CPU to offload jobs to PIM, all cores are seen as regular processors by the OS despite having different architecture and performance, and a program can run on either the main CPU, PIM cores, or both. The OS can migrate jobs between CPU and PIM cores.

Each PIM core has a private L1 cache, without L2 or shared caches (inherited from prior work [70] and also justified in Section III-B3). L1s are kept coherent using a low-overhead, software-assisted coherence protocol [34]. Specifically, shared read-write pages are not cacheable in PIM's L1s. When an application spans both CPU and PIM, they are not cacheable in CPU's LLC either. We also use the same dynamic classification mechanism in [34] to identify private/shared and read-only/read-write pages.

B. Methodology

1) Simulator: We modify zsim, a Pin-based simulator [63] to model the PIM-enabled system described above. Table I shows the detailed specification of the simulated system.

2) LC Applications: We study six diverse LC services (all that can be compiled successfully in simulation) from Tailbench [44]. We use the integrated configuration of Tailbench for easier simulation on zsim [44]. Request inter-arrival times follow an exponential distribution [48]. For each simulation, we instantiate the same number of application threads as the number of available cores, and run an application for 30 seconds of simulated time (about 72 billion cycles), including 20 seconds of warmup and 10 seconds of execution, except for Sphinx which we execute for 60 simulated seconds to collect enough requests due to its higher latency and lower request-per-second (RPS). The simulated execution time is consistent with prior work on these applications, using zsim [43, 44].

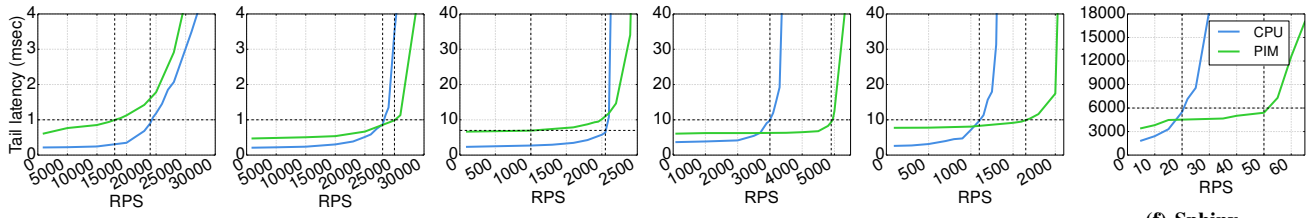
Fig. 2 shows tail latency, defined as the 99th percentile latency, with increasing RPS under CPU and two PIM stacks. The QoS is set as the knee of the CPU curve [21], marked with horizontal lines and recorded in Table II. We define *max load* of a service as the maximum RPS under QoS. The data in Table II is collected when running each application at *max load* on the main CPU. Unlike typical PIM targets, LC applications do not typically use high memory bandwidth.

3) Characterized Architectures: Compared to the main CPU, PIM stacks have *lower memory access latency*, since PIM cores are physically closer to memory; *shallow memory hierarchy*, due to the limited area of the logic die that may be better utilized with more cores rather than more cache; *wimpier cores*, due to the power and area constraints that cannot accommodate high-performance brawny cores.

To understand the impact of each of these factors brought on by PIM, we enumerate all combinations of them, to study eight different architectures, as listed in Table III, each with:

- **High/low memory latency**, i.e., with/without the off-chip portion of memory accesses from the CPU (the measured average memory access time is $65ns$/155 CPU cycles and $21ns$/50 CPU cycles, respectively).
- **Deep/shallow memory hierarchy**, i.e., with/without private L2 and shared LLC.
- **Brawny/wimpy cores**, detailed in Table I.

We start from a baseline CPU socket with 4 brawny cores and 8MB of LLC (the first architecture in Table III). Since LC applications have to run for at least a few tens of simulated seconds to get statistically meaningful tail latency, and simulation time grows super-linearly with core count, we characterize a relatively small system to keep simulation time manageable (1-4 days per simulation). Later in Section V-B3, we experiment with larger systems. We first study the impact of core type and memory hierarchy on the CPU die (the first four CPU-centric architectures with high memory latency in Table III). Keeping the same CPU die area, core count varies when core type or memory hierarchy changes. Since the area of one brawny core is roughly 2.5 wimpy cores or 2 LLC



(a) Silo (b) Masstree (c) ImgDNN (d) Xapian (e) Moses (f) Sphinx

Fig. 2: Tail latency with increasing input load (RPS) on the main CPU and PIM stacks. Horizontal lines show the knee of the CPU curve, which defines QoS. Vertical lines show *max load* (maximum RPS under QoS) achieved on the main CPU and PIM stacks.

TABLE II: Latency-critical applications

Application	Silo	Masstree	ImgDNN	Xapian	Moses	Sphinx
Domain	In-memory DB	Key-value store	Image recognition	Web search	Real-time translation	Speech recognition
Target QoS	1 ms	1 ms	7 ms	10 ms	10 ms	6 s
Per-core IPC	1.18	1.09	1.07	1.38	0.99	0.55
LLC MPKI	1.50	6.02	16.78	3.66	23.17	10.40
LLC Miss Rate	2%	12%	45%	37%	77%	47%
Memory Bandwidth (GB/s)	0.32	3.40	7.83	2.58	10.29	2.57
Memory Capacity (GB)	1.8	9.3	0.3	5.6	2.5	1.4

TABLE III: Impact of memory hierarchy, memory latency and core type. All the characterized architectures take up similar area, enforced by varying the core count. Architectures that could be realized on the main CPU or PIM are categorized as CPU-/PIM-centric. For each application, we evaluate its tail latency over the QoS target at low load (values larger than 1 represent QoS violations), and normalized max load (higher is better). For each cell, the darker the green/red, the better/worse. Memory latency in the last row is still low, but slightly higher than in Arch 5-8 due to the impact of data placement described in Section III-D.

	Characterized Architecture					Tail Latency/QoS Target							Normalized Max Load (Max RPS under QoS)						
	ID	**MemLat**	**Core**	**MemHie**	**#Cores**	**Silo**	**Masstree**	**ImgDNN**	**Xapian**	**Moses**	**Sphinx**	**AVG**	**Silo**	**Masstree**	**ImgDNN**	**Xapian**	**Moses**	**Sphinx**	**AVG**
CPU-centric	1	High	Brawny	Deep	4	0.22	0.21	0.33	0.37	0.26	0.26	0.28	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	2	High	Brawny	Shallow	6	0.36	0.24	0.28	0.40	0.35	0.29	0.32	0.89	0.85	1.17	1.33	1.41	1.25	1.15
	3	High	Wimpy	Deep	10	0.71	0.40	1.29	0.62	0.84	0.56	0.74	0.68	0.72	0.00	0.87	0.55	1.50	0.72
	4	High	Wimpy	Shallow	16	0.74	0.53	1.09	0.63	0.89	0.55	0.74	0.53	0.93	0.00	1.12	1.09	2.25	0.99
Un-realistic	5	Low	Brawny	Deep	4	0.21	0.18	0.25	0.36	0.22	0.23	0.24	1.08	1.09	1.20	1.05	1.23	1.20	1.14
	6	Low	Brawny	Shallow	6	0.28	0.20	0.20	0.37	0.27	0.27	0.26	1.05	1.00	1.66	1.50	1.82	1.50	1.42
PIM-centric	7	Low	Wimpy	Deep	10	0.59	0.34	0.97	0.59	0.67	0.54	0.62	0.89	0.85	0.71	1.13	0.91	1.75	1.04
	8	Low	Wimpy	Shallow	16	0.58	0.40	0.75	0.60	0.66	0.47	0.58	0.79	1.15	0.93	1.65	1.82	2.65	1.50
PIM	9	Low*	Wimpy	Shallow	8+8	0.61	0.47	0.95	0.61	0.77	0.49	0.65	0.68	1.09	0.63	1.58	1.50	2.50	1.33

slices,[1] when core type is changed from brawny to wimpy, core count is changed from 4 to 10 (or from 6 to 16) under deep (or shallow) memory hierarchy.

We then study the impact of low memory latency by (unrealistically) moving the CPU die to the memory side (i.e., using the logic layers in memory stacks). Architectures (Arch) 7&8 with low memory latency and wimpy cores could potentially be realized on one or more PIM stacks, and are therefore categorized as "PIM-centric". Arch 5&6 are unrealistic to prototype on either CPU or PIM (low memory latency and brawny cores cannot coexist), and are presented for completeness of the characterization study.

[1]Under 22nm process, one simulated CPU core and its private caches take roughly $13mm^2$, and one slice of LLC (2MB) is about $7mm^2$ [3]. The area of one simulated PIM core together with its private cache is conservatively estimated as $5mm^2$ under the same process [2], roughly a quarter of the total area of one CPU core and one slice of LLC.

Finally, we study Arch 9, a realistic PIM architecture that takes the area constraint of each PIM stack into account. Given the $50mm^2$ area budget of the logic layer in a PIM stack [70], the 16 wimpy cores have to be separated into two memory stacks, 8 cores each. This complicates memory page placement, and since cross-stack memory access increases memory latency (detailed in Section III-D).

C. Implications of the PIM Architecture

We use tail latency and max load to quantify the impact of PIM on LC applications, as shown in Table III. Tail latency is collected at low load (i.e., less than 10% of per-core utilization), and is normalized to the QoS of each application; a value over 1 represents a QoS violation, marked as red in the table. Max load is normalized to the load achieved in Arch 1. Tail latency at low load tells us if an application can run any load while meeting QoS, and if so, how low tail latency is.

Max load, on the other hand, shows the maximum throughput regardless of how low tail latency is, as long as QoS is met.

1) Impact on Tail Latency: Tail latency is always lower on brawny cores, regardless of memory latency or memory hierarchy. Among CPU-centric architectures, wimpy cores cause QoS violations for ImgDNN. This is consistent with prior studies that show the advantage of brawny cores in tail latency [22, 40], and consistent with current high-end servers that still widely adopt brawny cores [7].

However, when the cores are closer to memory, wimpy cores are also able to provide QoS guarantees. This is because when memory latency is lower, the need to hide memory latency is less critical, therefore the high issue width and the aggressive out-of-order mechanisms are not necessary. Shallow memory hierarchy increases tail latency for most applications due to the lack of LLC to exploit data locality, but lower memory latency helps reduce latency by up to 30% (average 16%).

2) Impact on Max load: As long as QoS is met, max load is a more important metric to quantify the performance of LC applications. From Table III, we find that:

- Arch 8, a PIM-centric architecture, on average outperforms all CPU-centric architectures. Masstree, Xapian, Moses and Sphinx achieve higher load on Arch 8 than on CPU-centric architectures. This advantage remains on Arch 9. This clearly shows the potential of PIM to many LC applications.
- Looking into Arch 8, applications' preference degree to PIM varies significantly: Masstree is only slightly better on PIM, while Sphinx achieves more than doubled max load on PIM.
- Comparing architectures that only vary in memory hierarchy, the one with shallow memory hierarchy is usually more appealing. This is in part because cloud services usually have poor data locality and large memory footprints that do not fit in the LLC [31, 65], as can be seen in the high LLC miss rate in Table II. The benefit of shallow memory hierarchy is larger when memory latency is low due to the smaller LLC miss penalty.
- Lower memory latency significantly improves max load, by up to 67% (average 26%), when comparing architectures that only vary in memory latency.

D. Impact of Data Placement

We have shown the potential of PIM architectures for LC applications: low memory latency, shallow memory hierarchy, and many wimpy cores provide the highest max load on average. When realizing the characterized PIM architecture (Arch 8) in real PIM (Arch 9), the 16 wimpy cores have to be separately placed in two memory stacks (shown in Fig. 1 and Table I), due to limited area in the logic layer of 3D memories. Consequently, memory pages may spread across stacks, i.e., memory accesses are no longer homogeneous: PIM cores accessing data from neighbor stacks have slightly higher access latency. This makes data placement critical to achieve the best performance; reducing cross-stack communication could maximize the benefits from low memory latency on PIM. Comparing with Arch 8, Arch 9 considers the increase of memory latency from cross-stack communication, and uses first-touch as its static page policy (Section III-D1).

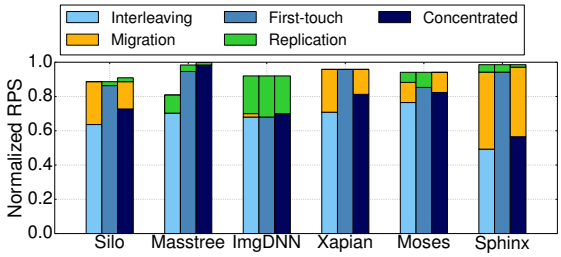


Fig. 3: Max load under various static (blue bars) and dynamic (yellow and green bars) data placement policies, normalized to an ideal policy with all local memory accesses (Arch 8 in Table III). Each application uses cores from 2 PIM stacks.

We study several widely used memory policies in NUMA systems and apply them to the PIM-enabled system, by essentially treating a PIM stack as a NUMA node. We define a memory access to be *local/remote* if the memory page requested is in the same/different stack as/from the core which initializes the memory request. We currently do not consider data layout and the heterogeneity of memory access latency within a single PIM stack (e.g., if each on-chip core were "closer" to a particular memory stack module), since the latency difference between different vaults in the same stack is almost negligible compared to the latency difference between PIM stacks and between CPU and PIM.

1) Static Data Placement: page location is decided upon its first access. There are three common static policies:

- **Interleaving (IL):** Interleave all pages across all nodes.
- **First-touch (FT):** Allocate in the node upon the first access.
- **Concentrated (CC):** Allocate in as few nodes as possible.

Under low load, when cores from a single node are enough, CC is the best static policy providing the most local memory accesses (memory bandwidth is far from saturation for these applications). However, the best policy changes with input load. When each application spans two memory stacks (Fig. 3), FT outperforms IL and CC by up to 46%, because under FT, private pages are always local to the core running the thread, i.e., all private pages have local memory accesses. Fig. 4 shows the memory page access breakdown by page type, including private/shared pages, and read-only/read-write pages. FT is 38% better than CC and 46% better than IL for Sphinx, which has over 95% of private page accesses.

2) Dynamic Data Placement: the location of pages can also be dynamically manipulated at runtime. Current OSes support:

- **Page migration** that migrates pages to other nodes, based on locality [72]. We explore migration for private pages.
- **Page replication** that replicates pages across nodes, usually limited to read-only pages to avoid synchronization overheads. We replicate only shared read-only pages.

Yellow and green bars in Fig. 3 show the benefit of adding page migration and replication on top of each static memory policy. Despite the non-negligible differences between the three static memory policies, they all perform similarly

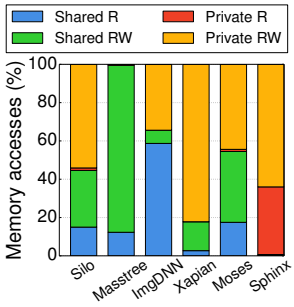


Fig. 4: Memory page access decomposition.

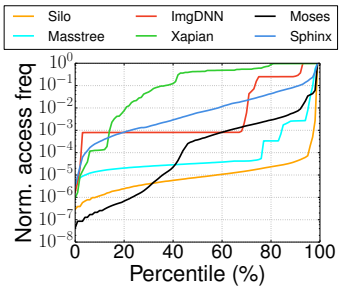


Fig. 5: Page access frequency normalized to max frequency per app.

TABLE IV: Symbol definition.

Symbol	Definition
N	Number of LC applications
M	Number of PIM stacks
C	Number of CPU cores
P	Total number of PIM cores (P/M cores/stack)
R_i	Max load ratio of a CPU core over a PIM core for app i (take ceiling if not divisible)
A_i=(c_i, p_i)	Core allocation of app i, with c_i CPU cores and p_i PIM cores
A =$\{A_i \mid i = 1..N\}$	Core allocation
$RPS(A_i)$	*max RPS* of app i under allocation A_i

after applying dynamic page migration and replication. Page migration provides up to 40% improvement for applications with mostly private pages, such as Sphinx. Page replication provides up to 20% improvement for applications with many shared read-only page accesses, such as ImgDNN.

E. Summary

The studies in this section offer two main takeaways:

- PIM has substantial potential to achieve higher performance than CPU. More than half of the characterized applications achieve better performance (i.e., max load) on PIM. Since applications have different preferences over CPU and PIM, it is critical to take such preferences into account during resource management; running an application on the wrong type of resource may cause QoS violations in the worst case.
- Data placement is critical to fully leverage the low memory latency provided by PIM, and dynamic page placement must be considered to achieve the best performance. Additionally, LC applications have fluctuating loads that result in frequent changes in core allocation, which in turn can lead to excessive page migrations/replications. Therefore, it is critical for data placement to (1) be aware of any change in core allocation to trigger dynamic page manipulation, and (2) reduce the amount/overhead of dynamic page manipulation.

IV. PIMCloud Design

Given the importance of managing cores and data placement for LC applications, we design PIMCloud, a QoS-aware resource manager for LC applications in PIM-enabled systems.

A. Design Principles

Resource management is trivial when there is only one application per node; it can simply take up all the resources to maximize its performance. A more common but challenging scenario is multi-tenancy, i.e., multiple cloud applications are colocated on the same node. Multi-tenancy is widely adopted in the cloud to improve server utilization [15, 29, 30], and the colocated applications may include one or more LC applications [21, 55, 59], each has its own QoS requirement. Given this scenario, PIMCloud follows three design principles:

- **QoS-aware** to meet the QoS of each co-scheduled LC application sharing a node. Batch jobs are of lower priority, and thus can take resources that are unused by LC applications. Being QoS-aware also indicates being adaptive, since the resource requirement of an LC application changes with time due to inevitably fluctuating loads.
- **PIM-aware** to manage PIM-introduced resources, including heterogenous cores, heterogenous memory latency and hierarchy, and data placement across stacks. Since differences in memory latency and hierarchy are encoded in the core type, i.e., a CPU/PIM core always comes with higher/lower memory latency and deep/shallow memory hierarchy, managing the heterogenous memory latency and hierarchy is encoded into our core management algorithm. As a byproduct, PIMCloud is applicable to heterogeneous systems with homogeneous memory like big.LITTLE.
- **Rapidly converging** to find feasible allocations fast by reducing the allocation space as much as possible. Long convergence time may lead to QoS violations and more inertia to adapt to load changes. Rather than aiming at the absolute optimal allocation at the expense of an exponentially increasing convergence time, it is preferred to find a *good enough* allocation as quickly as possible.

B. Core Allocation

We first introduce the intuition of core allocation in PIMCloud, aiming at largely reducing the core allocation space. The core allocation algorithm is detailed in Section IV-D.

Applications have different preferences with respect to core type (Section III-B3). To quantify this preference, for each colocated LC application i, we collect the maximum RPS under QoS (i.e., *max load*) when running the application on the main CPU and on all the PIM stacks (memory policy is first-touch), namely L_{CPU} and L_{PIM}. We then obtain R_i by $(L_{CPU}/C)/(L_{PIM}/P)$, and taking the ceiling if the result is not an integer. Intuitively, one CPU core provides the same performance as R_i PIM cores for application i (note that R_i can be less than one). A large R_i signals strong preference to CPU cores. Unless otherwise stated, we number applications in decreasing order of R, i.e., $R_1 \geq R_2 ... \geq R_N$.

An ideal resource manager with unlimited time to make decisions would exhaustively search the whole allocation space, where each application can be allocated any number of CPU and PIM cores. According to the stars and bars method [16],

there are $\binom{C+N}{N}\binom{P+N}{N}$ different allocations. The problem would be much simpler if all cores were identical: in that case, the problem would be reduced to choosing how many cores to allocate to each application, for which there are *only* $\binom{C+P}{N}$ possible assignments. For example, in a system with 4 CPU cores, 16 PIM cores, and 2 colocated applications, the exploration space can be reduced from 2,295 to 190 possible assignments. The difference also increases rapidly with more colocated applications. **The key to accelerating decision convergence is to organize the search such that the space looks as homogeneous as possible, while striving for an almost optimal assignment.**

The intuition behind our approach is to leverage the varying preference degree of different applications, to sort applications and cores such that cores are allocated to applications in order. Given $R_1 \geq R_2 ... \geq R_N$, we assign *only* CPU cores to applications $\{1, 2, \dots k-1\}$; *only* PIM cores to applications $\{k+1, k+2, \dots N\}$; both (or either) core types to application k. Note that, once we determine the optimal k and how many cores of each type application k gets, allocating cores within each application set $\{1, 2, \dots k-1\}$ and $\{k+1, k+2, \dots N\}$ becomes a homogeneous assignment problem, limited to deciding how many cores each application gets.

In actuality, our mechanism is even simpler: Beginning with application 1 , we determine how many CPU cores each application needs to satisfy QoS and proceed to allocate them. The first application that runs out of CPU cores and needs additional PIM cores to satisfy QoS is called application k'. Applications $k'+1$ through N receive PIM cores only. Note that k' may not be the optimal k above; this simplification is intentional. Also, depending on the circumstances, other orders might yield more efficient assignments. For example, if the majority of applications prefer PIM cores, it might be more optimal to allocate in reverse, from application N to 1. Or it might make sense to allocate according to how "strongly" applications prefer CPU or PIM cores (e.g., $R_i = 10$ and $R_j = 0.1$ both prefer one core type ten times over the other type). For simplicity, we systematically perform a 1-to-N assignment in all cases, which our evaluation shows that it yields significant gains. Any leftover cores can then be assigned to BE jobs.

We now provide the theoretical backup of the mechanism.

Definition 1. $A \leq A'$ if and only if
(1) $\sum_{i=1}^{N} c_i \geq \sum_{i=1}^{N} c'_i$, (2) $\sum_{i=1}^{N} p_i \geq \sum_{i=1}^{N} p'_i$, and
(3) $\forall i = 1..N,\ RPS(A_i) \leq RPS(A'_i)$.

Definition 2. A is suboptimal if $\exists A'$ such that $A \leq A'$.

Table IV includes all symbol definitions. Suboptimal allocations consume more cores, while producing less profit (lower max RPS under QoS). To identify suboptimal allocations, we start from $N = 2$. We find that in optimal allocations:

1. App 1 always gets fewer than R_1 PIM cores if it has not occupied all CPU cores, i.e., A is suboptimal if $p_1 \geq R_1$ and $c_2 > 0$. To prove this, we can construct another allocation A' by moving a CPU core from app 2 to app 1 in exchange for R_1 PIM cores. Formally, let $c'_1 = c_1 + 1, p'_1 = p_1 - R_1, c'_2 = c_2 - 1, p'_2 = p_2 + R_1$. The performance of app 1 remains the same under A'. App 2 gets R_1 more PIM cores at the cost of one CPU core. Since $R_1 \geq R_2$, it gets more PIM cores than needed to recover its performance. Therefore, $A \leq A'$.
2. Furthermore, the number of PIM cores of app 1 can be reduced to R_2, i.e., A is suboptimal if $p_1 \geq R_2$ and $c_2 > 0$. Formally, we can construct A' by setting $c'_1 = c_1 + 1, p'_1 = p_1 - R_2, c'_2 = c_2 - 1, p'_2 = p_2 + R_2$, such that $A \leq A'$.

In summary, the heterogeneity degree of app 1 is significantly reduced, from P to $min(R_1, R_2) = R_2$ PIM cores.

To generalize to N applications,

Theorem 1. *For any allocation A, define k as the last application that has CPU cores, i.e., $c_k > 0$ and $\forall i > k, c_i = 0$. A is suboptimal if $\exists x < k, p_x \geq R_k$.*

Proof. Construct $A' = (A'_1, A'_2, ..., A'_N)$ such that $A \leq A'$.

$$A'_i = \begin{cases} A_i & \text{if } i \neq x \text{ and } i \neq k \\ (c_i + 1, p_i - R_k) & \text{if } i = x \\ (c_i - 1, p_i + R_k) & \text{if } i = k \end{cases}$$

This means that, for any optimal allocation:

1) App k (*middle app*) separates all services into two classes. We call applications with larger R value *PIM-averse apps*, and those with smaller R value *PIM-friendly apps*.
2) For any *PIM-averse app*, its PIM cores are limited to a number smaller than its R value. The more colocated jobs, the lower the bound. Our characterization of Tailbench shows that $R \in [2, 8]$, with a median of 3. In practice, we find that performance stays almost the same when discarding a small number of PIM cores. This is because heterogeneous core assignment results in worse usage of LLC on the main CPU, as shared read-write pages are non-cacheable (Section III-A. This offsets the higher computation capabilities brought by the few PIM cores, and increases the allocation space. Therefore, PIMCloud disallows heterogenous cores for *PIM-averse apps*.
3) The *middle app* is the only one that may jointly have CPU cores and more than R_{mid} PIM cores. As it is the last service with CPU cores, if they are not enough, it may need to also get more PIM cores to sustain its input load.
4) *PIM-friendly apps* only get PIM cores.

PIMCloud Rule: All applications except for the *middle app* are assigned homogeneous cores. PIMCloud limits heterogeneity to only the *middle app*, and allocates only CPU cores to *PIM-averse apps*, and only PIM cores to *PIM-friendly apps*. This is thereafter referred to as the *PIMCloud rule*. Applying it both initially and at runtime significantly reduces the allocation space, making it the same as in homogeneous multicore systems. Note that allowing only a single application to span CPU and PIM may not be as beneficial in larger-scale systems with more colocated applications; this design decision is primarily made to make scheduling practical. However, we show later in Section V-B3 that PIMCloud still

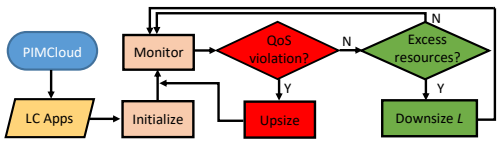


Fig. 6: Overview of the PIMCloud resource manager.

outperforms prior work in larger systems since it takes into account resource preference through lightweight profiling.

C. Data Placement

We select CC as the default static page allocation policy in the PIMCloud resource manager. Despite FT being the best static page policy (Section III-D1), CC is the best policy when cores are concentrated in one stack at low load, and the best policy with dynamic data migration/replication, when cores span multiple stacks at high load (Section III-D). We define *resident stack* of each application as the default memory stack for the OS to allocate memory pages for the application.

Dynamic data placement happens upon core reassignment of applications that are allocated PIM cores. When load increases and PIM cores from non-resident stacks are allocated, private pages need to be migrated from the resident stack to another stack, and shared read-only pages will be replicated. When load decreases and PIM cores from non-resident stacks are deallocated, private pages will be migrated back to the resident stack. Replications of shared read-only pages may be removed.

However, due to the limited memory capacity per stack and the application colocation, there may not be enough capacity in the target stack to hold all pages to be migrated/replicated. PIMCloud stops migrating/replicating when it runs out of memory. Therefore, it is necessary to identify "important" pages that benefit the most from dynamic data placement.

Fig. 5 shows the page access frequency distribution of each application. All services have a small fraction of pages that are accessed orders of magnitude more times than the rest. These hot pages dominate memory accesses, and are critical to be placed in the appropriate memory stack. Therefore, PIMCloud always starts migration/replication from the hottest pages.

D. PIMCloud Resource Manager

Fig. 6 shows an overview of the PIMCloud manager.

1) Initialization: For each LC application, PIMCloud initializes a core allocation and a resident stack (Section IV-C). This initialization tries to spread out resources to reduce resource contention, while following the PIMCloud rule.

First, PIMCloud conducts a quick profiling step to obtain the preference degree to CPU of each application. Profiling is only triggered when a new/previously-unseen LC application enters the system. A load generator is needed to generate representative user requests at any given rate. During profiling, it collects max RPS (not under QoS) for each LC application on (a) one CPU core and (b) one PIM core, by injecting sufficient requests over 500 ms in each case (1s in total), and noting the achieved RPS. Since it takes much longer to collect *max load*, PIMCloud approximates the preference degree by taking the max RPS ratio. Verification by preference degree using *max load* (collected offline) shows that the order of applications remains unchanged. LC applications are then sorted in decreasing order of their preference degree.

Algorithm 1: *Downsize(A)* to reclaim excess resources from application A.

```
while Slack[A] > Threshold[A] do
    goodSlack = Slack[A];
    adjustCore(A, -1);   // Remove a core from app A.
    if removed core is PIM-remote then
        migrate private pages (of the thread previously running on the
          removed core) back to A's resident stack;
        if no more PIM core allocated in that stack then
            remove all replicated pages in that stack;
    monitor latency till latency stabilizes or slack < 0;
    if slack[A] < 0 then
        revert();
        Threshold[A] = goodSlack;
        monitor latency till latency stabilizes;
```

Applications are partitioned into $M+1$ groups, such that each group has $\lceil N/(M+1) \rceil$ applications. The first group is scheduled to CPU cores with stack M being their resident stack, and the $i^{th} (i \geq 2)$ group is scheduled to cores in stack $i-1$, which is also their resident stack. Applications in the same group equally partition the available CPU/PIM cores. If a resident stack i runs out of memory, the OS will start allocating memory from stack $i+1$, then $i+2$, ..., M, 1, ...

A BE pool is also initialized to save all unallocated cores and memory capacity from LC applications, which can be used for BE jobs if they exist, or powered off to save energy.

2) Performance Monitoring: PIMCloud continuously monitors request latency every 100ms [21]. Less frequent monitoring can reduce tail latency jitter, but also delays the convergence of resource adjustments.

3) Latency Slack: This captures the distance of tail latency from a QoS target [21, 51]. Slack<0 represents a QoS violation, while a large positive slack signals excessive resources. To compute slack, latency is monitored for 5 consecutive 100ms intervals [21]. The median tail latency of the five intervals is used to compute the slack. This smoothens out short latency spikes that are not due to lack of resources.

E. Resource Adjustment

PIMCloud *upsizes* to counteract a QoS violation, and *downsizes* to reclaim excessive resources.

1) Downsize(A): gradually moves A's cores to the BE pool (Algorithm 1). Pages may be migrated/de-replicated. Latency is then monitored in 100ms intervals until it stabilizes or a QoS violation occurs, which triggers a revert of the downsize operation. We use average rather than tail latency to monitor stability, as it is more accurate in signaling the latency trend. Core reassignment takes 2-3 intervals to stabilize, while page manipulation takes 3-8 intervals, due to the overhead of page manipulation, and the higher inertia until it reflects on an application's overall latency [43].

A downsize threshold is maintained for each application, representing the least positive slack an application can sustain, i.e., further reducing resources would result in a QoS violation. A slack larger than the threshold signals opportunities to reclaim excess resources (e.g., when load decreases). The downsize threshold is initialized to 0, and is updated after every failed *downsize* or successful *upsize* operation.

2) Upsize(A): shifts resources to applications with QoS violations (Algorithm 2). Resources are obtained lazily and eagerly. In the former case, *upsize* iterates through the application list, shifting as many resources as possible to the next job via *downsize*. Eventually excessive resources will move to problematic jobs. In the eager case, it will also proactively reclaim resources from the remaining applications.

If a remote PIM core is allocated, depending on the idle memory capacity in the remote memory stack, pages will be migrated/replicated in decreasing order of their access frequency. If there is not enough memory capacity, PIMCloud could benefit from aggressively swapping cold pages of other applications in the remote stack. However, this causes severe memory fragmentation; an application may end up with pages spread across many stacks. This causes more page movements when load drops, and complicates the management of data placement. Therefore, only idle memory space is leveraged. A thread is scheduled to run on the newly obtained remote PIM core. This thread will not be context-switched with other threads to avoid previous private pages becoming shared pages, increasing memory access latency.

When the system is oversubscribed, PIMCloud communicates with the cluster-wide scheduler to trigger admission control, so that some requests/applications are redirected to another machine to reduce the system load. Note that PIMCloud is a per-node resource manager, not a cluster-wide job scheduler. We leave the interaction between these two to future work. Once the QoS violation of application A is resolved, its downsize threshold is updated to the current slack (minimum positive slack), to avoid aggressive downsizing in the future.

3) adjustCore(A, count): adjust $count$ cores for A. The PIMCloud rule should still be satisfied afterwards.

When $count = +1$, A will always try to obtain a CPU core first. This is regardless of A's preference because we always run out of CPUs before going to PIM cores. Since applications are already sorted in decreasing order of their preference for CPUs, if there is a CPU core left, all previous applications have no PIM cores allocated (i.e., if a prior application has PIM cores, this means that CPU cores are unavailable). If a CPU core is not available, A is allocated PIM cores. PIMCloud will try securing a PIM core on A's resident stack if possible, or a remote PIM core if otherwise. For any core type, it checks the BE pool first, and then applications with smaller R value than R_A. If several applications can supply the desired type of core, it chooses the application with the largest latency slack.

When $count = -1$, one core is moved from application A to the BE pool. A CPU core will be removed if available. It not, a remote PIM core (not in its resident stack) will be removed, or a local PIM core if not available again.

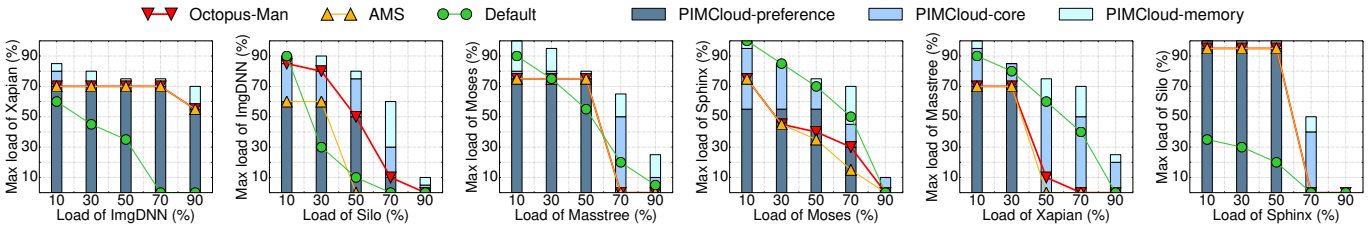
Algorithm 2: *Upsize* to resolve QoS violations.

```
for each application A do
    if Slack[A]>Threshold[A] then
        downsize(A);
        give released resources to the next application;
    else
        while Slack[A]<0 do
            adjustCore(A, +1);   // Obtain a core for app A.
            if Failed to obtain a core then
                admissionControl();   // System oversubscribed.
            else
                if obtained core is PIM-remote then
                    migrate private pages (of the thread scheduled
                      to run on the obtained core) to the remote
                      stack;
                    if this is the first remote core in that stack then
                        replicate shared read-only pages to the
                          remote stack;
            monitor latency till latency stabilizes;
            if Slack[A] > 0 then
                Threshold[A] = Slack[A];
```

4) Application churn: When an application terminates and exits, its allocated resources are recycled to the BE pool. When a new job arrives, PIMCloud reranks all applications based on their R-value. Suppose the new job is in rank i; it will take one core from job in rank $i + 1$ if $i \neq n$; otherwise, it will take one core from job in rank $i - 1$. This ensures that the PIMCloud rule holds. PIMCloud will then choose the stack with the most available space as the application's resident stack. If this initial allocation does not meet QoS, PIMCloud will adjust the resources to resolve any QoS violations.

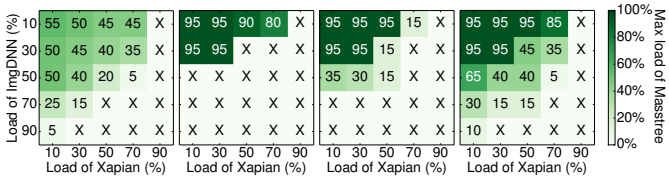
5) PIMCloud Overhead: Existing OSes provide interfaces for dynamic thread pinning (e.g., *taskset* in Linux, 100us/adjustment) and page manipulation (1us/4KB page or 250ms/1GB). This overhead of adjusting core allocation and/or adjusting page placement is reflected in PIMCloud's varying monitoring time to wait for latency stabilization after any resource adjustment (Section IV-E1).

To monitor page access frequency, a new field representing the access frequency is added to each page table entry. It is incremented for every memory access, and is reset periodically to avoid overflow. Not being on the critical path of request processing, maintaining page access frequency does not affect application performance. We compare the tail latency of each application with and without such frequency tracking, and don't observe any latency difference. To ease the process of page migrations and replications, we additionally implement a special `syscall` that takes two integers n, k as input, and returns the n most-accessed memory pages from stack k, together with their page type (private/shared, read-only/read-write). The `syscall` returns results under 10ms, negligible compared to the actual page manipulation.



(a) ImgDNN+Xapian (b) Silo+ImgDNN (c) Masstree+Moses (d) Moses+Sphinx (e) Xapian+Masstree (f) Sphinx+Silo

Fig. 7: Colocation of 2 LC applications. Y-axis is the maximum RPS (as a percentage of *max load*) of the second application when the first application is at a given RPS (x-axis) and both applications meet QoS. The three competing policies are shown in curves, while PIMCloud is shown in bars, broken down to identifying core preferences (PIMCloud-preference), dynamic core adjustment (PIMCloud-core), and dynamic data placement (PIMCloud-memory).



(a) Default (b) AMS (c) Octopus-Man (d) PIMCloud

Fig. 8: Colocation of Xapian, ImgDNN and Masstree. Each cell represents the max RPS (as a percentage of *max load*) of Masstree when Xapian and ImgDNN run at given RPS (x and y axes), and all three applications meet QoS. Crossmarks mean that QoS cannot be met concurrently.

V. Evaluation

A. Methodology

We use the same simulated system and applications as described in Section III-B. Each application is instantiated with a fixed 20 threads. We focus our evaluation on single machine experiments, since PIMCloud is a per-node resource manager, and can be installed on every node in a large cluster/datacenter. For distributed applications spanning multiple nodes, the system's cluster manager handles load balancing across nodes hosting the same application, and PIMCloud manages the resources of each node's application instances.

We compare against three resource managers:

- **Default** relies on the OS to manage resources. Threads are mapped to cores in a round-robin fashion. Cores are chosen randomly during context switches. *First touch* (FT) is the static memory policy, with no dynamic page manipulation (same for the other two resource managers below).
- **AMS [70]** is designed for batch applications in PIM systems. It profiles cache miss ratio curves leveraging hardware support (not available on commodity servers), to identify applications' preferences to PIM. It then schedules threads one at a time, according to their preferences.
- **Octopus-Man [60]** manages LC applications in heterogeneous multicores, like ARM big.LITTLE [24]. Since it optimizes for energy efficiency, it always starts from small cores, and then switches to big cores until QoS is met. As discussed in Section II-B1, Octopus-Man assumes a single LC service per node, and thus does not consider any application ordering during resource allocation. We compare against an optimized version of Octopus-Man, which "magically" enumerates all possible application orderings, and selects the best without incurring any overheads.

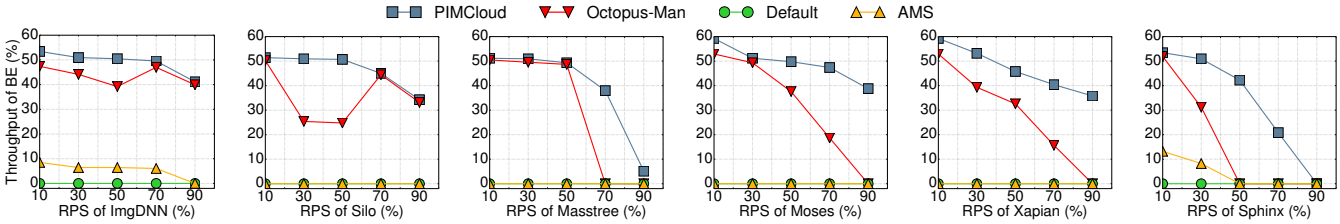
We first evaluate colocation under constant loads. Due to too many possible application mixes (i.e., 15/30/15/6 2-/3-/4-/5-app mixes), we select a few representative ones but still cover a diverse set of colocation scenarios. Assume applications are sorted by their preference degree to CPU cores, we choose 6 2-app mixes: (A_1, A_5) and (A_2, A_6) with strong complementary preferences; (A_3, A_5) and (A_4, A_6) with slight complementary preferences; (A_1, A_2) and (A_3, A_4) with similar preferences. We also make sure that every application appears an equal number of times across all mixes, i.e., each application appears precisely twice in the six 2-app mixes. We also choose a 3-app combination (A_1, A_3, A_5) that has applications with a strong, medium, and weak preference for CPU cores, and a 6-app mix (A_1, A_2, ..., A_6) with all studied applications.

For each mix, we sweep the load of each service from 5% to 100% of their respective *max load* (i.e., the maximum RPS under QoS when running alone on all available cores), in 5% load increments. For each run, we warm up for 20 seconds of simulated time (48 billion cycles), and then run each resource manager until it converges, i.e., it either finds an allocation without QoS violations, or determines that no viable allocation exists that meets QoS. We record all load combinations that are able to meet QoS with each resource manager.

B. Constant Load

1) Colocation of 2 LC Applications: Fig. 7 shows performance across six diverse 2-app mixes. The three competing policies are shown in lines. The Y-axis shows the maximum RPS of the second application when the first application is at a given RPS (x-axis) and both applications meet QoS. PIMCloud is broken down to three components shown as stacked bars. PIMCloud-preference shows the result when allocating only the preferred type of cores to each application, based on their R values. Dynamic core adjustments and data placement are shown as PIMCloud-core and PIMCloud-memory.

Default is not aware of core heterogeneity, and fails to allocate the right type of cores to applications with strong preference, such as ImgDNN, Silo and Sphinx. Therefore,



(a) ImgDNN+Xapian **(b) Silo+ImgDNN** **(c) Masstree+Moses** **(d) Moses+Sphinx** **(e) Xapian+Masstree** **(f) Sphinx+Silo**

Fig. 9: Colocation of 2 LC and 1 BE applications. The first LC application is at a given RPS, and the second one is at 10% of its max load. Y-axis is throughput of the BE application when both LC applications meet their QoS targets.

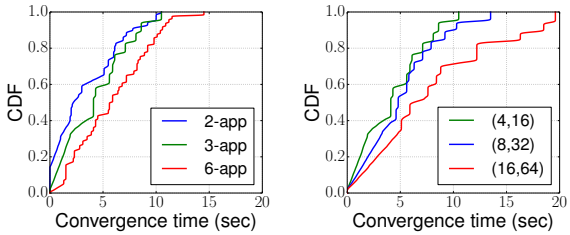
Default performs the worst in mixes with these applications (Fig. 7a, 7b, and 7f). PIMCloud-preference alone outperforms *Default* by being aware of the preference of each application.

AMS identifies each application's core preferences successfully most of the time, using cache miss ratio curves.However, it is not designed for LC applications whose core requirements vary with input load. AMS assumes a fixed number of cores for each application, which works only for throughput-oriented jobs. As shown in Fig. 7, the curves for AMS are almost always flat, as each application occupies one type of cores regardless of the load of the other application. Moses+Sphinx (Fig. 7d) is the only exception. According to their miss ratio curves, both of them prefer PIM stacks, and thus are both scheduled to PIM cores(CPU cores are always idle).

Octopus-Man is not preference-aware: it always tries small cores first, and if not meeting QoS, it tries big cores. Because we evaluates an optimal version of Octopus-Man by enumerating all possible application orderings, the best ordering is usually the preference-aware one, similar to AMS. In addition, it also does not explore heterogeneous core allocations, and thus performs similarly to AMS in many cases.

PIMCloud outperforms all three policies. First, PIMCloud is preference-aware, and outperforms *Default* when applications have strong preferences over core type. Second, PIMCloud dynamically adjusts core allocations, and allocates heterogeneous cores to one of the colocated applications, outperforming AMS and Octopus-Man when one application is at high load. Finally, PIMCloud handles dynamic data placement, allowing the system to sustain up to 30% (average 9.5%) additional load under QoS. Finally, regarding effective machine utilization (EMU) [51], which captures the total load under QoS of all co-scheduled jobs, *Default*, AMS, Octopus-Man, and PIMCloud achieve on average 89%, 93%, 97%, and 121% EMU, respectively. PIMCloud achieves up to 80%, 80% and 70% (average 32%, 28% and 24%) higher EMU than *Default*, AMS, and Octopus-Man, respectively. Dynamic core adjustment and data placement each contribute up to 60% and 30% higher EMU (average 17% and 10%).

2) Colocation of 3 LC Applications: PIMCloud works for any number of colocated LC applications. Resource allocation becomes more critical with more colocated jobs. Fig. 8 shows colocation of the 3-app mix. Compared with *Default*, PIMCloud improves max load for Masstree by 5-40% when applications are at varying load points. Following the cache



(a) # Applications **(b) System size**

Fig. 10: PIMCloud Scalability.

miss ratio curves, AMS schedules Masstree to CPU cores, and ImgDNN and Xapian to PIM cores. When the aggregate load of ImgDNN and Xapian increases, the static allocation of AMS does not meet QoS. Octopus-Man outperforms AMS as it allocates CPU cores when PIM cores are insufficient. PIMCloud is able to further push the "pareto frontier" to the bottom right, meeting QoS under for more load combinations. In summary, PIMCloud achieves up to 50%, 85% and 70% higher EMU than *Default*, AMS, and Octopus-Man.

3) Scalability: PIMCloud can be applied to any number of colocated applications at any system scale (i.e., any number of CPU cores and PIM stacks), despite the evaluation above is done on a relatively small system. We now study the scalability of PIMCloud with respect to the number colocated applications and the system size using convergence time, i.e., the time required for PIMCloud to find a feasible allocation without any QoS violations, or conclude that no viable allocation exists. The overhead of the online profiling is one second per application before allocation starts. Fig. 10a shows the CDF of convergence time for 2-, 3- and 6-app mixes, and Fig. 10b shows that of the 3-app mix with 4/16, 8/32, and 16/64 CPU/PIM cores. Convergence time varies from zero time, when the initial allocation achieves QoS, to 10.2 and 10.6, 16 seconds for 2-, 3- and 6-app mixes respectively. The max convergence time increases sub-linearly with more colocated applications and larger systems, despite the fact that, theoretically, the allocation space increases exponentially. In more than 50% of cases, PIMCloud converges in 3, 5 and 8 s for 2-, 3-, and 6-app mixes respectively. For 16 CPU cores, PIMCloud still converges in 6 s more than 50% of the cases, with the maximum convergence time being under 20 s.

4) Colocation with BE applications: We also evaluate PIMCloud with the presence of best-effort (BE) jobs running in the background, by colocating one BE and two LC applications. We construct a CPU and memory intensive 20-threaded BE job using microbenchmarks [28], which achieves higher throughput when utilizing all PIM cores. This mimics data-intensive applications that tend to favor PIM [49, 53, 61].

Fig. 9 shows the BE throughput, when the first LC application (LC_1) has varying load, and the second LC application (LC_2) is fixed at 10% of max load. *Default* does not provide core isolation, while AMS does not isolate cores within CPU or PIM. AMS schedules one application to either CPU or PIM. Since there are three applications, at least two of them will be scheduled to the same type of cores, leading to contention. The lack of core isolation leads to BE taking most resources, and LC services violating their QoS under *Default* and AMS. Octopus-Man allocates cores exclusively for each application, outperforming *Default* and AMS significantly. Compared to PIMCloud, BE throughput is usually lower under Octopus-Man, as Octopus-Man is oblivious of core preference, and does not allow heterogeneous core assignment when one core type is insufficient for applications at high load. Octopus-Man always tries to allocate PIM cores to each LC application, and the BE job tends to get the less preferable CPU cores. PIMCloud identifies the preferences of each LC application, and frees up more PIM resources for the BE job. When LC_1 has strong preference to CPU and is at high load (Fig. 9a and 9b), Octopus-Man achieves similar results to PIMCloud. Achieved BE throughput is even higher than when LC_1 is at 10-50% of load. This is because Octopus-Man allocates PIM cores to LC_1 at low load, leaving the undesirable CPU cores for the BE job. However, as the load increases, PIM cores are insufficient. Octopus-Man will then schedule the LC service to CPU cores, freeing up more PIM cores for the BE job, resulting in higher throughput. Due to better page management, BE throughput is still higher with PIMCloud.

C. Fluctuating Load

PIMCloud is designed to handle various load patterns in LC applications. To evaluate dynamic load, we colocate two LC applications, Moses and Xapian. These two applications do not have strong preference over core type, so they favor *Default*. We keep Xapian at constant load, and gradually increase and then decrease the load of Moses, to simulate a diurnal load pattern [21]. Fig. 11 shows tail latency under *Default* and PIMCloud, and resource allocations over time.

PIMCloud starts by placing Moses on CPUs and Xapian on one PIM stack, allocating stack1 and stack0 as their residence stacks, respectively. When both applications are at low load, they are downsized to save energy. At 2 s, removing a CPU core causes a QoS violation for Moses, which is quickly resolved after yielding the core back. At 10 s, PIMCloud detects a QoS violation for Moses which signals a load increase, so the core allocation is increased. Moses experiences a latency spike every time load increases, but tail latency quickly recovers after PIMCloud adjusts its allocation. PIMCloud performs

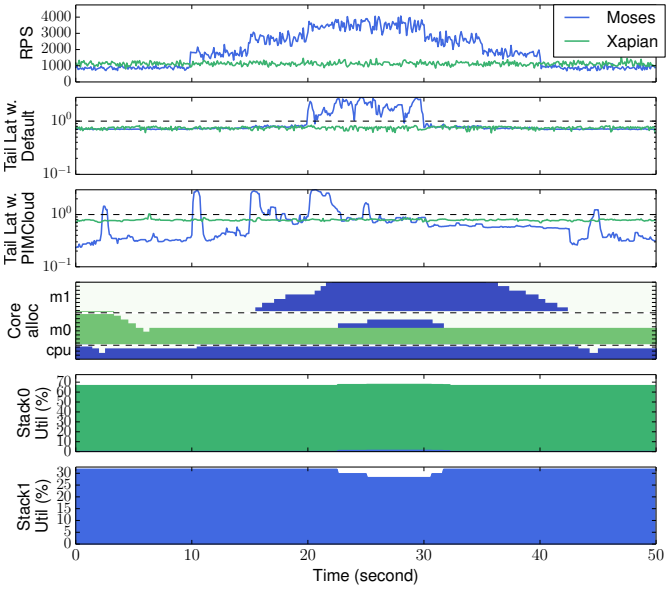


Fig. 11: Latency and resource allocations with PIMCloud under constant load for Xapian, and varying load for Moses. Latencies are normalized to their respective QoS.

page migration and replication for Moses every time a remote core is allocated, at 23 s and 25 s. After 30 s, PIMCloud downsizes Moses due to the large latency slack. However, under *Default*, Moses experiences long-lasting and severe QoS violations in $t = [20, 30]\ s$, when Moses is at high load.

VI. Conclusions

We have proposed PIMCloud, a PIM-aware resource manager for cloud environments that dynamically adjusts the resource allocations of colocated latency-critical applications to satisfy QoS. We show that PIMCloud improves effective machine utilization by up to 70% and 85% (average 24% and 33%) under 2-app and 3-app mixes, compared to state-of-the-art managers, and adjusts successfully under fluctuating load.

Acknowledgement

This work was supported in part by NSF and the Semiconductor Research Corporation (SRC) through the DEEP3M Center, part of the E2CDA program; and by DARPA and SRC through the CRISP Center, part of the JUMP program. Christina Delimitrou was partially supported by a Sloan Foundation Research Award, a Microsoft Research Faculty Fellowship, an Intel Rising Star Award, a Google Faculty Research Award, and NSF grants NeTS CSR-1704742 and CCF-1846046. The authors wish to thank the COE/CIS/Tech Information Technology Support Group, in particular Michael Woodson, for their technical assistance.

REFERENCES

[1] "Cortex-A57 overview," https://developer.arm.com/ip-products/processors/cortex-a/cortex-a57.

[2] "Cortex-A57, wikichip," https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a57.

[3] "Haswell, wikichip," https://en.wikichip.org/wiki/intel/microarchitectures/haswell_(client).

[4] "High bandwidth memory (hbm) dram," https://www.jedec.org/standards-documents/docs/jesd235a.

[5] "Hmc specification 2.0," Hybrid Memory Cube Consortium.

[6] "Intel Xeon processors," https://www.intel.com/content/www/us/en/products/processors/xeon.html.

[7] "Server market share in q3 2020," https://www.extremetech.com/computing/318217-amd-arm-both-increased-their-server-market-share-in-q3-2020.

[8] "ThunderX2 ARM-based processors," https://www.marvell.com/server-processors/thunderx2-arm-processors/.

[9] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in International Symposium on Computer Architecture (ISCA), 2015.

[10] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in International Symposium on Computer Architecture (ISCA), 2015.

[11] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), 2015.

[12] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems," in 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016.

[13] O. O. Babarinsa and S. Idreos, "Jafar: Near-data processing for databases," in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD), 2015.

[14] A. Barbalace, A. Iliopoulos, H. Rauchfuss, and G. Brasche, "It's time to think about an operating system for near data processing architectures," in Proceedings of the 16th Workshop on Hot Topics in Operating Systems, 2017.

[15] L. Barroso and U. Hoelzle, The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Synthesis lectures on computer architecture, 2013.

[16] P. Billingsley, Probability and measure. John Wiley & Sons, 2008.

[17] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2018.

[18] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "Scheduling and page migration for multiprocessor compute servers," in Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1994.

[19] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017.

[20] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2016.

[21] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.

[22] S. Chen, S. GalOn, C. Delimitrou, S. Manne, and J. F. Martínez, "Workload characterization of interactive cloud services on big and small server platforms," in International Symposium on Workload Characterization (IISWC), 2017.

[23] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[24] H. Chung, M. Kang, and H.-D. Cho, "Heterogeneous multi-processing solution of exynos 5 octa with arm® big. little technology," Samsung White Paper, 2012.

[25] J. Cong and B. Yuan, "Energy-efficient scheduling on heterogeneous multi-core architectures," in International Symposium on Low Power Electronics and Design (ISLPED), 2012.

[26] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on numa systems," in Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2013.

[27] J. Dean and L. A. Barroso, "The tail at scale," Communications of the ACM, vol. 56, no. 2, pp. 74–80, 2013.

[28] C. Delimitrou and C. Kozyrakis, "iBench: Quantifying interference for datacenter workloads," in Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC). Portland, OR, September 2013.

[29] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2013.

[30] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2014.

[31] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.

[32] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, Y. He, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems," in Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 2019.

[33] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in International Conference on Parallel Architecture and Compilation (PACT), 2015.

[34] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT), 2015.

[35] M. Gao and C. Kozyrakis, "HRL: Efficient and flexible reconfigurable logic for near-data processing," in International Symposium on High Performance Computer Architecture (HPCA), 2016.

[36] V. Gavrielatos, A. Katsarakis, A. Joshi, N. Oswald, B. Grot, and V. Nagarajan, "Scale-out ccNUMA: Exploiting skew with strongly consistent caching," in Proceedings of the Thirteenth EuroSys Conference, 2018.

[37] B. Gu, A. S. Yoon, D. Bae, I. Jo, J. Lee, J. Yoon, J. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A framework for near-data processing of big data workloads," in 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[38] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, O. Mike, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in gpu systems," in ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[39] S. Hurkat and J. F. Martínez, "VIP: A versatile inference processor," in International Symposium on High-Performance Computer Architecture (HPCA), 2019.

[40] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid, "Web search using mobile cores: Quantifying and mitigating the price of efficiency," in Proceedings of the 37th Annual International Symposium on Computer Architecture, 2010.

[41] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for μsecond-scale tail latency," in 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI}), 2019.

[42] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in 42nd Annual International Symposium on Computer Architecture (ISCA), 2015.

[43] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict QoS for latency-critical workloads," in Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 2014.

[44] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in IEEE International Symposium on Workload Characterization (IISWC), 2016.

[45] J. S. Kim, D. S. Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies," BMC genomics, vol. 19, no. 2, p. 89, 2018.

[46] Y. C. Kwon, S. H. Lee, J. Lee, S. H. Kwon, J. M. Ryu, J. P. Son, O. Seongil, H. S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H. S. Shin, J. Kim, B. Phuah, H. Kim, M. J. Song, A. Choi, D. Kim, S. Kim, E. B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, and N. S. Kim, "25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications," in IEEE International Solid- State Circuits Conference (ISSCC), 2021.

[47] J. H. Lee, J. Sim, and H. Kim, "Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models," in International Conference on Parallel Architecture and Compilation (PACT), 2015.

[48] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in Proceedings of the 9th European Conference on Computer Systems, 2014.

[49] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018.

[50] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3: energy-efficient microservices on smartnic-accelerated servers," in {USENIX} Annual Technical Conference ({ATC}), 2019.

[51] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA), 2015.

[52] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations," in Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture (MICRO), 2011.

[53] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in International symposium on high performance computer architecture (HPCA), 2017.

[54] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell, "Hipster: Hybrid task manager for latency-critical cloud workloads," in International Symposium on High Performance Computer Architecture (HPCA), 2017.

[55] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-agent task management for colocated latency-critical cloud services," in International symposium on high performance computer architecture (HPCA), 2020.

[56] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2014.

[57] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "The case for RackOut: Scalable data serving using rack-scale systems," in Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC), 2016.

[58] S. Panneerselvam and M. Swift, "Rinnegan: Efficient resource use in heterogeneous architectures," in Proceedings of the 2016 International Conference on Parallel Architectures and Compilation Techniques (PACT), 2016.

[59] T. Patel and D. Tiwari, "CLITE: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in International symposium on high performance computer architecture (HPCA), 2020.

[60] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang, "Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers," in IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015.

[61] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads," in International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014.

[62] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozych, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in Proceedings of the Third ACM Symposium on Cloud Computing (SoCC), 2012.

[63] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA), 2013.

[64] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2017.

[65] A. Shahab, M. Zhu, A. Margaritov, and B. Grot, "Farewell my shared llc! a case for private die-stacked dram caches for servers," in International Symposium on Microarchitecture (MICRO), 2018.

[66] K. Skadron, Y. Xie, J. F. Martínez, S. Swanson, and J. Patel, "CRISP: Center for Research in Intelligent Storage and Processing in memory," in Government Microcircuit, Applications, and Critical Technology Conf. (GOMACTech), 2018.

[67] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2000.

[68] A. Sriraman and T. F. Wenisch, "μ suite: a benchmark suite for microservices," in International Symposium on Workload Characterization (IISWC), 2018.

[69] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "ReQoS: Reactive static/dynamic compilation for QoS in warehouse scale computers," in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2013.

[70] P.-A. Tsai, C. Chen, and D. Sánchez, "Adaptive scheduling for systems with asymmetric memory hierarchies," 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 641–654, 2018.

[71] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in 39th International Symposium on Computer Architecture (ISCA), 2012.

[72] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on CC-NUMA compute servers," in Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1996.

[73] S. Wang, Y. Liang, and W. Zhang, "Poly: Efficient heterogeneous system and application management for interactive applications," in International Symposium on High Performance Computer Architecture (HPCA), 2019.

[74] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "SMiTe: Precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers," in Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2014.