2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Criticality-Aware Instruction-Centric Bandwidth Partitioning for Data Center Applications

Liren Zhu, Liujia Li, Jianyu Wu, Yiming Yao, Zhan Shi†, Jie Zhang,
Zhenlin Wang‡, Xiaolin Wang, Yingwei Luo and Diyu Zhou
Peking University, †Huawei, ‡Michigan Tech
{zhuliren, liujia_li, jywu, yim, jiez, wxl, lyw, diyu.zhou}@pku.edu.cn,
shizhan11@hisilicon.com, zlwang@mtu.edu

***Abstract*—To reduce operational costs, modern data centers co-locate high-priority latency-critical (LC) tasks and low-priority best-effort (BE) tasks on the same physical node to increase resource utilization. However, such co-location leads to contention for memory bandwidth, resulting in priority inversion, where BE tasks severely slow down LC tasks. This priority inversion often leads to violations of the quality of service (QoS) requirements for LC tasks, defeating the purpose of co-location. Prior approaches to this issue either fail to enforce the QoS requirements for LC tasks or underutilize memory bandwidth.**

We present PIVOT, a novel bandwidth partitioning system that overcomes the limitations of prior approaches based on two key insights. First, memory accesses from LC tasks must be prioritized across *all* the components on the memory path rather than a single component, as done in prior work. Second, only the scheduling of a selective portion of performance-critical loads (i.e., those causing a long stall on the re-order buffer), instead of all memory accesses from LC tasks, should be prioritized. To leverage these insights, PIVOT overcomes the key challenge of accurately identifying performance-critical loads while incurring minimal runtime overhead by proposing a two-phase profiling technique. Our extensive evaluation shows that PIVOT improves effective machine utilization by up to 34.5% while increasing the throughput of the BE applications by up to 2.76× compared to state-of-the-art approaches.

I. INTRODUCTION

Modern data centers suffer from high operational costs, mainly due to underutilized resources. Major industry practitioners report that the core utilization rate can be less than 50% for 90% of the operational time [18], [30]. Worse still, these high operational costs are exacerbated by the ever-increasing expenses of procuring and maintaining hardware [70].

To minimize operational costs, a common strategy is to improve resource utilization by co-locating best-effort (BE) tasks (e.g., big data analytics) with latency-critical (LC) tasks (e.g., web search engines) on the same physical node [17], [24], [45], [56], [70]. The service level on the BE tasks is not guaranteed, so they should only be scheduled if resources permit. Unlike BE tasks, LC tasks ultimately interact with human users and thus often come with Quality-of-Service (QoS) requirements that must be met. Indeed, prior study [6] reports that even an increase of 100 milliseconds of service time results in millions of dollars lost for companies like Amazon. Therefore, the scheduling of LC tasks should be prioritized over BE tasks.

However, naively co-locating LC and BE tasks often leads to priority inversion caused by contention for shared hardware resources, such as memory bandwidth. Specifically, BE tasks are often memory intensive, issuing a large number of memory accesses, and thereby blocking those issued by LC tasks. Such blocking slows down the execution of LC tasks, increases their response latency, and ultimately violates the QoS requirements, resulting in severe economic loss.

Existing mechanisms for such a priority inversion issue suffer from severe limitations. Intel Memory Bandwidth Allocation (MBA) [8] throttles the bandwidth usage of BE tasks by aggressively inserting additional delays before the Last-Level Cache (LLC). While this approach enforces QoS requirements, it significantly underutilizes memory bandwidth (wasting up to 62% as shown in Figure 2) and thus defeats the purpose of task co-location. ARM Memory Partitioning and Monitoring (MPAM) [3] prioritizes scheduling the memory access requests issued by the LC task at the memory bandwidth controller [72]. However, our evaluation finds that MPAM cannot enforce the QoS requirements of LC tasks under heavy bandwidth contention (§II-B). In summary, neither existing mechanism resolves the inherent tension between enforcing QoS requirements and maximizing bandwidth utilization, and thus, they still cannot effectively reduce operational costs.

We present PIVOT, a novel and effective mechanism to prevent priority inversion caused by bandwidth contention. PIVOT overcomes the limitations of prior approaches by resolving the aforementioned inherent tension with two key insights: noitemsep,nolistsep

- First, to enforce QoS requirements, **memory accesses from LC tasks must be prioritized across *all* the components on the memory path**. This is because, under heavy bandwidth contention, the queuing of memory accesses in one component can propagate to upstream components, eventually causing queuing in all the components. Therefore, scheduling memory accesses in a single component, as prior approaches do, is insufficient, since the memory access can be blocked in *any* component.
- Second, to resolve the inherent tension, **only the scheduling of a small portion of *performance-critical load instructions*, which causes a long stall on the re-order buffer (ROB), should be prioritized**. This is because, with out-of-order execution, a long ROB stall implies that

other instructions are likely to depend on this load. Hence, prioritizing the scheduling of such loads enables both themselves and all dependent instructions to retire faster, thereby effectively enabling the LC tasks to meet the QoS requirements. Furthermore, prioritizing scheduling a small portion of loads instead of all memory accesses as prior approaches do incurs minimal interference with the default scheduling algorithm and avoids underutilizing memory bandwidth.

To leverage these insights, a key challenge PIVOT must overcome is to *accurately* identify the performance-critical loads. Naively collecting the ROB stall cycles for all loads incurs prohibitive performance overhead. Instead, PIVOT proposes a novel *two-phase* profiling technique. The first phase is offline (i.e., conducted in a non-production environment), where PIVOT co-locates an LC and a BE stress task to identify loads that are highly unlikely to be critical under most scenarios (e.g., a load accessing a memory address accessed only a few instructions earlier) in the LC task. PIVOT achieves this by collecting, for the LC task, the ROB stall cycles for *all* the loads to form a set of *potential* performance-critical loads. Due to the high runtime performance overhead, this offline profiling lasts for 30 minutes, but it only needs to be performed once for each LC task. As a result, the long profiling time does not cause a problem since it takes data centers, on average, at least a few days to release new task [38], [53].

The second phase is conducted in the production environment where PIVOT identifies the actual performance-critical loads in the LC task by collecting the ROB stall cycles for only *the loads in the potential set*, thereby minimizing performance overhead.

We carefully design the memory access scheduling mechanisms in PIVOT to achieve multiple goals simultaneously. First, for each component on the memory path, PIVOT introduces a *priority queue* for performance-critical loads, ensuring their prioritized scheduling and preventing delays due to insufficient queuing space. In addition, PIVOT effectively prevents starvation of BE tasks while also handling cases where more than one LC task is co-located on the same physical node.

We implemented PIVOT on the Gem5 simulator [11] with about 6.2K lines of code. Our extensive experiments demonstrate that, since PIVOT overcomes the key limitations in prior hardware mechanisms, it even outperforms state-of-art hardware-software co-design approaches under various scenarios, improving the effective machine utilization [45] and the throughput of BE applications by up to 34.5% and 2.76×, respectively, while not causing QoS violation.

In summary, this paper makes the following contributions: noitemsep,nolistsep

- **Analysis.** We thoroughly analyze the existing bandwidth partitioning approaches and reveal their major limitations.
- **Novel insights.** We resolve the inherent tension between enforcing QoS requirements and maximizing memory bandwidth utilization by (1) enforcing access priority across the full memory path; and (2) only prioritizing the scheduling of performance-critical loads.
- **PIVOT.** We design PIVOT with two novel techniques: the two-phase profiling and scheduler for memory access.

PIVOT is open-source, available at https://github.com/TELOS-syslab/Pivot.

II. BACKGROUND AND MOTIVATION

This section first presents the problem PIVOT targets: memory bandwidth contention during task co-location (§II-A). Next, it discusses the limitations of existing approaches for addressing memory bandwidth contention (§II-B).

A. Opportunities and Challenges in Task Co-Location

Modern data centers often suffer from low resource utilization, leading to significant economic losses. For example, major industry partitioners such as Microsoft [18] and Alibaba [30] report that their core utilization rate is between 20% and 50% for as much as 90% of the time. This low resource utilization forces data center owners to procure additional hardware, significantly increasing operational costs [45], [70].

To improve resource utilization and thus minimize operational costs, data centers co-locate Latency-Critical (LC) tasks with Best-Effort (BE) tasks on the same physical machines [45], [47]. LC tasks, such as web search engines, key-value stores [26], interact with end human users and, as a result, are often associated with *Quality-of-Service (QoS) requirements*. QoS requirements must be met since violations lead to significant economic loss. For example, as underscored in several reports [6], users abandon video playback after waiting for just three seconds. In addition, even an increase of one hundred milliseconds of service time results in millions of dollars lost for Amazon [6]. As a result, the resource usage of LC tasks must be prioritized. Unlike LC tasks, BE tasks run to completion without interacting with end users. Thus, BE tasks can and should be scheduled only *when the resource permits*. Some examples of BE tasks are big data analytics, graph analytics, and large-scale scientific computing [1], [26], [28], [30], [45].

However, naively co-locating tasks may, in turn, defeat the purpose of cost-saving due to contention for shared resources [45], [47], [62], [63]. This paper targets an essential form of such contention: **memory bandwidth contention** [17], [70]. Memory bandwidth contention occurs because BE tasks are often memory-intensive; thus, during execution, their frequent memory accesses clog the memory access path. As we show in §II-B and reported by major industry practitioners [70], such clogging blocks the memory accesses issued by LC tasks, leading to QoS violations, thereby resulting in significant economic losses.

B. Limitations in Existing Approaches

A common approach to addressing memory bandwidth contention is to leverage the bandwidth partitioning mechanisms available in modern servers, such as Intel Memory Bandwidth Allocation (MBA) [8] and ARM Memory Partitioning and Monitoring (MPAM) [3]. Both mechanisms allow the software to specify each software thread's expected bandwidth usage

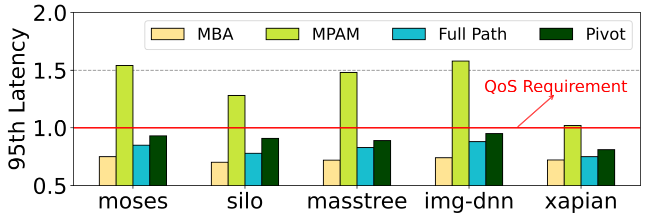


Fig. 1: Normalized 95th percentile latency of the LC tasks (lower is better). A bar higher than the red line is a violation of the QoS requirements.

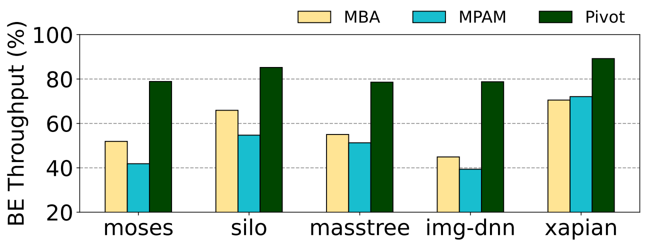


Fig. 3: Max normalized throughput of iBench (with respect to the throughput of 7-thread iBench running alone) when there is no QoS violation for LC tasks (higher is better).

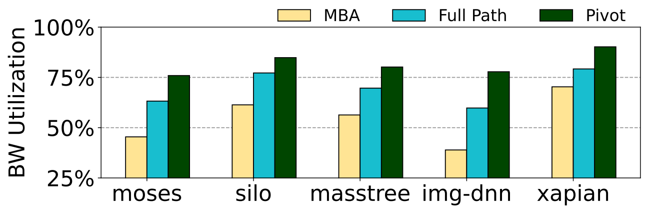


Fig. 2: Memory bandwidth utilization of different approaches (higher is better).

ratio. During execution, these mechanisms monitor and adaptively throttle the bandwidth usage of each thread.

The key difference between Intel MBA and ARM MPAM is their approach to throttling memory bandwidth usage. Intel MBA enforces memory bandwidth limits by introducing a controller between the L2 cache and the Last-Level Cache (LLC). If a software thread overuses memory bandwidth, the controller inserts additional delays into its memory accesses. In contrast, ARM MPAM enforces memory bandwidth usage by modifying the memory bandwidth controller module between the memory bus and the memory controller. MPAM assigns each software thread a priority value based on how much the thread under- or overutilizes the expected bandwidth. If memory access requests from different threads are queued at the memory bandwidth controller module, MPAM prioritizes scheduling those issued by threads with higher priority.

Unfortunately, both memory bandwidth partitioning mechanisms suffer from significant limitations. To demonstrate this, we used an iBench application [20] with seven threads as a memory-intensive BE task. Each thread in iBench sequentially copies the content from one private 64MB buffer to another. For MBA, we set the expected memory bandwidth of the LC task to **minimal** values that meet the QoS requirements. For MPAM, we set the expected memory bandwidth usage of the LC task to be **100%** of the overall bandwidth. Additionally, for both mechanisms, we partitioned the LLC to reserve the maximum possible space for the LC task, preventing contention in the LLC from affecting its performance. We used five different LC tasks. Following prior work [17], we set the QoS requirement by identifying the knee of the load-latency curve and drove the LC task with 70% of the maximal load (See Figure 12). We conducted the experiments with Gem5 [11]. Section V details our experimental setup and results with other BE tasks.

Figure 1 shows that, even with 100% expected bandwidth usage, MPAM fails to enforce the QoS requirements for the LC task. This is because, as explained in §III-A, enforcing memory access priorities merely at the memory bandwidth controller is insufficient. While MBA successfully enforces the QoS requirements, it is overly conservative and significantly underutilizes memory bandwidth, using only 38.9% to 70.3%, as shown in Figure 2. As shown in Figure 3, such low bandwidth utilization reduces the throughput of BE tasks, defeating the purpose of minimizing operational costs through task co-location. In contrast, PIVOT successfully enforces QoS requirements and achieves the highest bandwidth usage and throughput for *all* workloads, as shown in the figures.

III. DESIGN INSIGHTS BEHIND PIVOT

This section presents our two key insights that enable PIVOT to overcome the limitations of existing approaches (i.e., inability to enforce QoS requirements and underutilization of memory bandwidth, as discussed in §II-B).

The starting point of PIVOT'S design is MPAM, since we believe that prioritizing memory access requests, as done by MPAM, is fundamentally better at utilizing memory bandwidth than delaying memory access requests, as done by MBA.

A. Enforcing Access Priority on Full Memory Path

To effectively enforce QoS requirements under bandwidth contention, our key insight is that **memory access priority must be enforced across *all* components on the memory path.** This is because, as shown in Figure 4, the root cause of QoS violations is that memory access requests issued by BE tasks block those issued by LC tasks in *multiple* shared memory system components (MSCs), including the L2 cache interconnect, coherent memory bus, memory bandwidth control module, and memory controller. MPAM schedules memory requests only at the memory bandwidth controller. This is insufficient because, under bandwidth contention, the queuing in the memory bandwidth controller propagates to upstream MSCs, eventually causing queuing in all shared MSCs. Therefore, the queues in multiple MSCs should be viewed as a ***single*** virtual queue, and thus memory requests issued by LC tasks should be prioritized across ***all*** these queues.

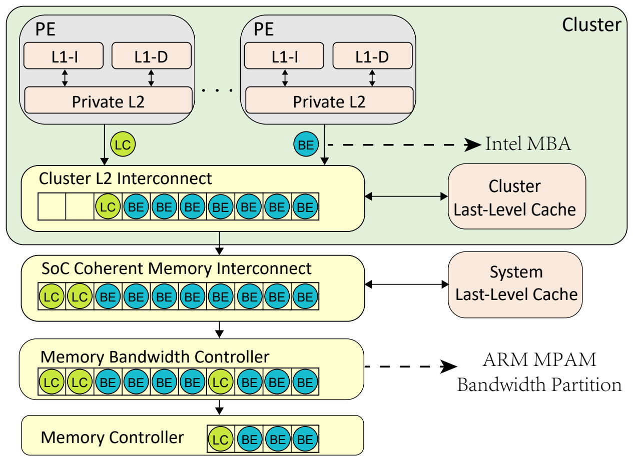


Fig. 4: Root cause of QoS violations. QoS violations occur because memory access requests issued by BE tasks block those issued by LC tasks in multiple shared components. Hence, scheduling memory access requests at a single component, as done by MBA and MPAM, is insufficient.

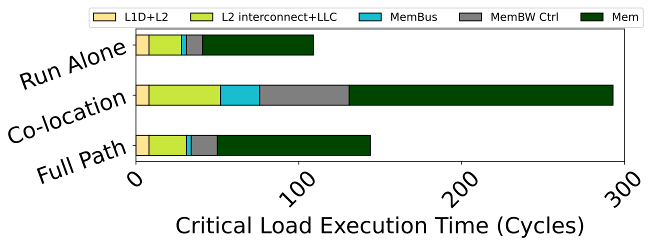


Fig. 5: A split of execution cycles of critical loads in Masstree, an LC task, on different memory system components. **Run Alone:** Masstree runs on its own. **Co-location:** Mastree runs together with a BE task. **Full path:** Mastree runs together with a BE task when full-path prioritization is enabled.

Figure 5 and Figure 6 evaluates the effectiveness of our insight using same setup as in Section II-B. As shown in Figure 5, with full path prioritization, the critical loads significantly reduce their waiting time in each component, almost matching the Run Alone case. Hence, after enforcing memory access priority across the full memory path, all LC tasks meet their QoS requirements even with 9 BE threads. Additionally, the tail latency increases slowly with the increasing number of threads, with a maximum increase of 22.3% (14.2% on average) from 1 to 9 BE threads. Finally, as shown in Figure 7, QoS violations always occur even if only one of the MSCs does not enforce access priority.

Insight #1. To meet QoS requirements, memory access priority must be enforced across all components on the memory path.

B. Enforcing Memory Access Priority at Instruction-Level

In addition to meeting the QoS requirements, PIVOT must maximize bandwidth utilization. To demonstrate the challenge

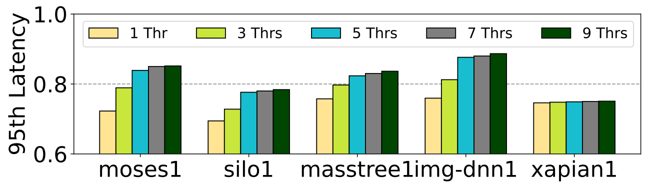


Fig. 6: Normalized 95th% latency of the LC tasks (with respect to QoS requirements) with varying numbers of BE task threads, achieved by enforcing access request priority across the full memory path (lower is better).

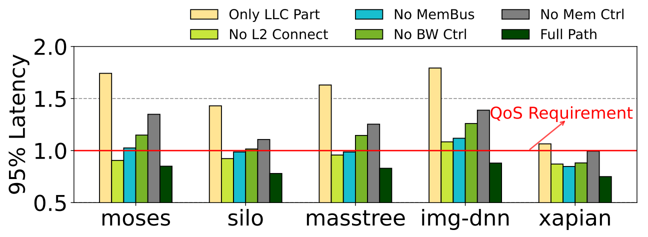
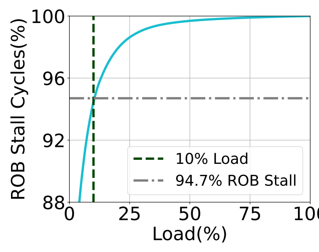


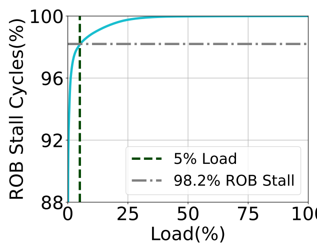
Fig. 7: Normalized 95th percentile latency of the LC tasks when a component does not enforce memory access priority (lower is better). A bar higher than the red line is a violation of the QoS requirements.

behind this, we enhance MPAM by enforcing the access priority on full memory path (Referred to as Full Path afterward). However, as shown in Figure 2, while Full Path performs better than MBA, it remains too conservative, utilizing only 59.8% to 79.2% of the overall memory bandwidth. This low utilization rate occurs because prioritizing LC memory accesses often conflicts with each component's default scheduling algorithm, aiming to maximize memory bandwidth utilization. For example, without the need to enforce memory access priority, the memory controller can prioritize scheduling requests that hit the row buffer, thereby increasing bandwidth utilization. As a result, the more memory accesses that are prioritized, the lower the memory bandwidth usage.

To resolve the tension between memory bandwidth utilization and enforcing QoS requirements, exploiting the fact that modern servers perform **out-of-order execution**, our second key insight is that **only the scheduling of performance-critical memory accesses instructions, specifically the load instructions causing a long stall on re-order buffer (ROB), should be prioritized.** Prioritizing such a selective portion of instructions is **most cost-effective** in minimizing response latency for LC tasks for the following two reasons. First, a long stall indicates that the load instruction misses CPU caches and accesses the main memory, demanding prioritized scheduling in the memory path. Second, and more importantly, a long stall implies that this load instruction does not depend on other instructions (since the out-of-order execution does not hide the long stall), but rather, other instructions are likely to depend on it. As a result, prioritizing the scheduling of such instructions



(a) Silo



(b) Moses

Fig. 8: Cumulative distribution of load instructions and reorder buffer (ROB) stall cycles for Silo and Moses.

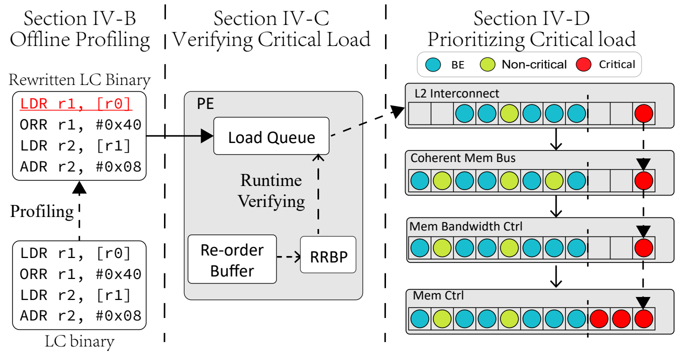


Fig. 9: Overview of PIVOT.

of LC not only enables them but *all* the dependent instructions to retire faster, thereby effectively minimizing latency. As an example, consider a key-value store such as Redis [2], the load instruction that obtains the address of the value is one such critical instruction; once this load instruction returns, subsequent dependent instructions, such as those that obtain the value from the address, can also retire.

The above insight immediately frees PIVOT from scheduling *all store instructions*. This is because modern servers use write buffers to optimize store performance. As a result, store instructions rarely cause stalls, and, for subsequent instructions, the processor is often able to find the dependent value in the write buffer. Therefore, enforcing access priority on store instructions provides little benefit for meeting QoS requirements.

In addition, as demonstrated in Figure 8, we found that the performance-critical loads constitute a very small portion of the overall loads. We omitted the results for other LC tasks since they are similar. The figure shows that fewer than **10%** of the load instructions cause over **95%** of the reorder buffer stall cycles. Thus, prioritizing the scheduling of these performance-critical instructions is unlikely to reduce bandwidth utilization significantly.

Insight #2. Prioritizing the scheduling of performance-critical load instructions effectively resolves the tension between enforcing QoS requirements and maximizing bandwidth utilization.

IV. PIVOT DESIGN

This section presents PIVOT, a novel, lightweight, and general solution for memory bandwidth contention based on the two key insights presented in §III. The novelty of PIVOT comes from the following aspects. First, based on Insight #1, PIVOT departs from prior approaches by scheduling memory accesses across all MSCs. Second, based on Insight #2, unlike prior approaches that associate access priority to each ***thread***, PIVOT associates access priority to each ***instruction***, thereby enabling scheduling at a much finer granularity.

We first provide an overview of PIVOT (§IV-A), followed by the design of each individual component (§IV-B, §IV-C, §IV-D), and conclude by discussing PIVOT's hardware implementation and the associated cost (§IV-E).

A. Design Overview

Deployment scenario. PIVOT targets *warehouse-scale data centers*, where each workload can be classified into BE and LC tasks, and the QoS requirements of LC tasks can be clearly specified. Moreover, the data center environment is suitable for *offline* profiling and binary rewriting, as demonstrated by the recent success of feedback-driven optimization [9], [14], [38], [39], [43] and post-link optimization [34], [35], [42], [53], [54] in data center. As detailed later, PIVOT similarly leverages these techniques to overcome a key design challenge.

Design goals and the key challenge. The overarching goal of PIVOT is to minimize data center operational costs by resolving the tension between enforcing QoS requirements and maximizing bandwidth utilization. In addition, we design PIVOT to meet the following goals. First, PIVOT aims to be general; its design relies solely on a few commonly available hardware features (e.g., performance monitoring counters, ROB), enabling practical and wide deployment. Second and more importantly, PIVOT is lightweight, incurring minimal performance and storage overhead to align with its overarching goal of minimizing operational costs.

The key design challenge in PIVOT is accurately identifying the performance-critical loads (§III-B). A naive approach is to collect the ROB stall cycles of all load instructions, but this incurs prohibitive performance overhead. We also consider an alternative design in which PIVOT identifies performance-critical loads by randomly sampling load instructions. However, given the small portion of performance-critical loads, this low-accuracy approach is highly unlikely to work well.

Key components and their workflow. Figure 9 illustrates the three main components and workflow of PIVOT. To overcome the challenge mentioned above, PIVOT employs a *two-phase* profiling approach.

The first phase is offline (i.e., not in a production environment), only needs to be performed once for each LC task and is used to exclude loads in the LC task that are highly unlikely to be critical under most scenarios (e.g., a load accessing a memory address accessed only a few instructions earlier). Specifically, in this phase, PIVOT runs the LC task with a stress BE workload that consumes lots of LLC sizes and memory bandwidth. This BE workload can be the same for all the LC tasks; our evaluation uses a simple memory copying workload for all the LC tasks, as detailed in §V-B. With this, PIVOT identifies a set of potential performance-critical loads in the LC task by recording, among other information, the ROB stall cycles of all loads. We estimate that this phase incurs a performance overhead of 75×, making the profiling time around 30 minutes for each LC task. This profiling time is acceptable since this is a one-time task and is still much smaller than the time for data centers to release new tasks, which on average takes at least several days [38], [53].

The second phase is online (i.e., in the production environment) and identifies the actual performance-critical loads in the LC task based on real user requests and co-located BE tasks. This phase incurs minimal performance overhead, as PIVOT only collects the ROB stall cycles for loads within the potential set of that LC task. Subsequently, PIVOT prioritizes the scheduling of these actual performance-critical loads across all shared MSCs.

B. *Offline Profiling*

Design. Figure 10 presents the offline profiling mechanism in PIVOT. PIVOT takes as input the binary of the target LC task, a user-specified stress BE task, and a few user-provided parameters that specify the criteria for determining whether a load is performance-critical in the LC task. The offline profiler outputs a rewritten binary of the LC task that flags potential performance-critical loads. This is possible since PIVOT introduces an extra bit in the instruction format to record the criticality of the loads.

The offline profiling works by first running the LC and the stress BE tasks on the same physical node and collecting runtime information of the LC task (Step ❶). Specifically, the profiler collects, for each load in the LC task, the LLC miss rate, the ROB stall cycles, and the number of executions. The profiler achieves this by utilizing hardware performance monitoring units [4], [65] and instruction tracing supports [5], [40]. Specifically, before each load, the profiler starts the performance counters for ROB stall cycles i.e., *topdown_be_bound.reorder_buffer* in Intel machines, *rob_stall* in ARM servers, and upon completion of the load, it records the value in memory.

Subsequently, in Step ❷, PIVOT decides whether a load is potentially performance-critical using three user-provided parameters: (a) the minimal execution frequency (i.e., the number of executions relative to all loads), with a default of 0.5%; (b) the minimal LLC miss rate, with a default of 10%, and (c) the minimum threshold for ranking in the top percentage of ROB stall cycles, with a of 5%. These default values are empirically set to maximize the chance of including actual performance-critical loads. PIVOT flags all loads that fall below the minimal execution frequency as normal loads since such low frequency makes them insignificant to the performance of the LC task. Next, PIVOT flags all loads as potentially critical if their LLC miss ratio exceeds the minimal threshold or if they rank higher in ROB stall cycles than the minimal ranking. Finally, in Step ❸, PIVOT marks the potential

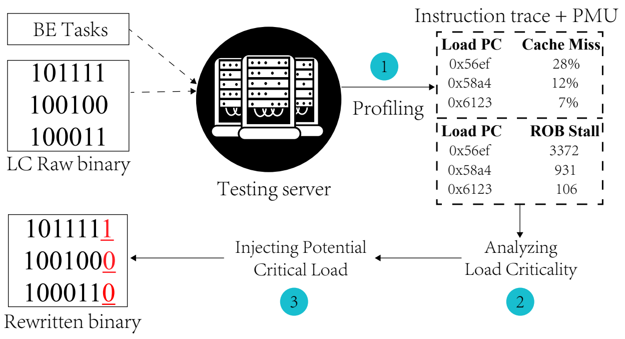


Fig. 10: The offline profiling in PIVOT.

critical load instructions by rewriting the binary to set the extra bit for each such instruction.

Discussion. We measured the overhead of offline profiling, which is around 75× (Figures omitted due to space limitations). Most of the overhead is due to starting and stopping PMU counters to collect ROB stall cycles (Step ❶), which, on average, take around 1200 cycles. Following prior work [21], [22], PIVOT profiles each LC task with a 20-second workload, which means the offline profiling phase of PIVOT is around 30 minutes. The length of the profiling time is acceptable, as discussed earlier §I. In addition, since the goal is to exclude loads that are highly unlikely to be critical, its effectiveness is not sensitive to the user-specified parameters, as shown in §VI-C.

C. *Identifying Real Performance Critical Loads Online*

At a high level, the online validation mechanism works as follows. For each potentially performance-critical load, PIVOT measures its ROB cycles. If a load instruction causes `long ROB stalls` (i.e., exceeding the access time of the LLC) for more than a specified number of times, PIVOT flags it as actual performance-critical. This threshold is determined by the current bandwidth usage of LC tasks. If the bandwidth usage is lower than the user-specified expected bandwidth, PIVOT sets a low threshold to include more instructions from the potential set aggressively. This thus helps PIVOT prioritize scheduling more loads and utilize additional memory bandwidth to meet QoS requirements. Otherwise, if the bandwidth usage is higher, PIVOT sets a higher threshold.

Figure 11 shows the online validation mechanisms of PIVOT. PIVOT introduces a new data structure called the Rutime ROB Block Predictor (RRBP) table, which tracks the number of times a load instruction causes `long ROB stalls`. The RRBP table is directly mapped, and load instructions are indexed by their addresses. To minimize the storage overhead, the RRBP table does not store tags. As a result, if two load instructions map to the same entry, they share the same counter. However, this potential inaccuracy is acceptable because performance-critical loads are infrequent, making such conflicts rare. Since LC tasks might operate in different phases with varying performance characteristics, PIVOT clears the

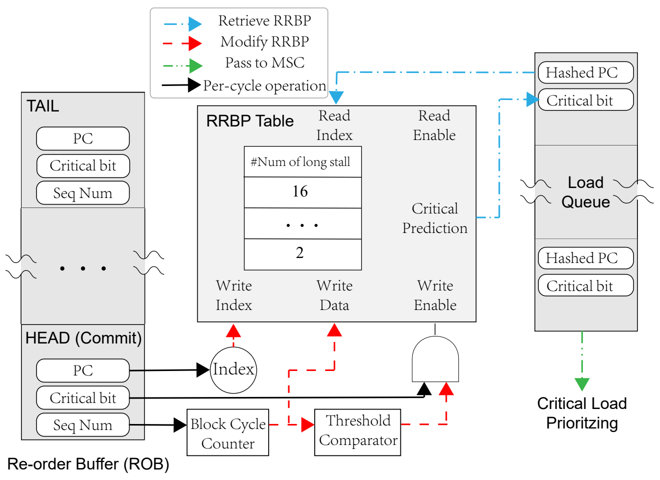


Fig. 11: The online profiling techniques of PIVOT.

RRBP table every 1M cycles (See §VI-C for evaluation on more refreshing intervals).

To flag the real performance-critical loads, whenever a load enters the load queue, PIVOT reads from the RRBP table and compares the number of `long ROB stalls` to the abovementioned threshold. PIVOT tags critical access requests by introducing a critical bit in the memory access requests. Since modern architectures can issue multiple loads in a single cycle, the RRBP table is designed with multiple read ports.

PIVOT counts the ROB stall cycles as follows. When a load arrives at the head of the ROB, PIVOT first checks if the instruction is potentially performance-critical by inspecting the critical bit in its instruction format. If so, PIVOT saves its sequence number into a private register and starts the `block cycle counter`. Then, in each cycle, PIVOT compares the current sequence number of the load with the saved one. If they match, PIVOT increments the cycle counter. If they do not match, the load instruction has been retired, and PIVOT then checks if this is a long stall. If it is, PIVOT increments the counter of the relevant entry in the RRBP table.

D. Prioritizing Scheduling Performance-Critical Loads

PIVOT prioritizes the scheduling of performance-critical loads by introducing a *priority queue* for each MSC. The priority queue only holds the memory access requests from performance-critical loads, while normal memory access requests still go into the ordinary queue. Each MSC prioritizes scheduling the requests in the priority queue whenever possible, thereby preventing normal memory access instructions from blocking performance-critical loads. Compared to an alternative design where each MSC queues the performance-critical loads as usual and re-orders them, the priority queue has the key advantage of preventing blocks due to a lack of space in the queue.

Another key design consideration in PIVOT is to prevent the starvation of BE tasks. PIVOT addresses this by implementing a maximum waiting cycle (8,000 DRAM cycles for memory controllers and 100,000 cycles for others) for normal memory access requests. If a request exceeds this wait time, PIVOT reorders it to the head of the queue.

Finally, to manage scenarios where multiple LC tasks are co-located on the same physical node , within the normal or priority queues, PIVOT utilizes MPAM (§II-A and §IV-E)) to schedule memory accesses based on both current and expected bandwidth usage. The effectiveness of this approach in co-location scenarios is evaluated in Section §VI-A2 and §VI-A3.

E. Hardware implementation

We implemented PIVOT on Gem5 [11] with 6,183 lines of code. To prevent task interference due to contention for LLC, PIVOT reuses the existing mechanisms to partition the LLC [3], [50].

Performance and storage overhead. In terms of performance overhead, accessing the RRBP table costs only a few cycles, comparable to an L1 cache access.

In terms of storage overhead, as discussed in §IV-C, PIVOT adds the following components to each PE: an 8-bit register for saving the sequence number, a 5-bit register for storing the index for the RRBP table, and an 8-bit comparator (for a 192-entry ROB), totaling $8 + 5 + 8 = 21bits$ In addition, PIVOT adds an extra bit to each entry in the ROB to flag potential criticality, resulting in a total of $192bits(192entries \times 1bit)$. The RRBP table consists of 64 entries, each with 6 bits to store the number of long stalls, leading to $64 \times 6 = 384bits$. Furthermore, PIVOT adds to each entry in the load queue (consisting of 64 entries) 1 bit to flag the actual criticality bit, and 6 bits to store the program counter (PC) index, totaling $64 \times 7 = 448bits$. In summary, the total extra storage for each PE is $21 + 192 + 384 + 448 = 1045bits$.

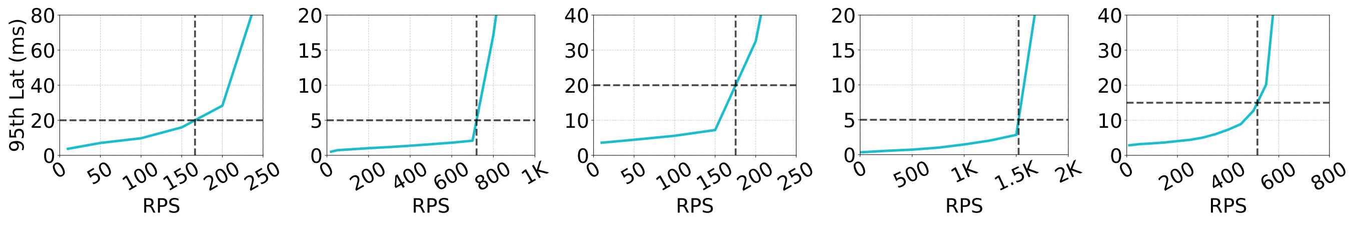
MPAM implementation. We could not find an existing MPAM implementation in Gem5, so we developed one based on the MPAM standard [3]. Our implementation modifies the memory bandwidth control module to introduce three priority classes: high (if the thread consumes more than the allowed maximal bandwidth), medium, and low (if the thread consumes less than the allowed minimal bandwidth). The monitoring window for bandwidth usage is 100,000 cycles, following Kunpeng 920 [66].

V. EXPERIMENTAL METHODOLOGY

This section describes the simulation infrastructure, machine model, and benchmarks used to evaluate PIVOT.

A. System Architecture Overview

Table II and Table III show the parameters of the target CPUs simulated in Gem5. Since many modern ARM servers now support the MPAM specification, such as NVIDIA Grace [25], ARM Neoverse [57], Marvell Thunder [61], and Huawei Kunpeng [66], we select Huawei Kunpeng and ARM Neoverse as the models to match our simulator parameters. The evaluation results in section VI-A, section VI-B, and section VI-C are based on the parameters in Table II. Additionally, Section VI-D shows the effect of Pivot on ARM Neoverse cores using parameters in Table III



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

Fig. 12: Tail latency with increasing input load on the Gem5 simulator. Horizontal lines show the knee of the curve, which defines QoS. Vertical lines show *max load* (maximum RPS under QoS).

Since the simulation time of the Gem5 simulator is proportional to the number of simulated processors, we simulate up to 8 cores running the data center applications to control the time consumed by the simulation (about 2-10 days per simulation). In the experiment, PARTIDs (an identifier to flag different threads for resource allocation) are assigned based on CPU serial numbers, ensuring that each CPUID corresponds to a unique PARTID. Each CPU executes a single thread.

B. Applications

Table I shows our workload from Tailbench [37], CloudSuite [1], and iBench [20]. Our stress BE task is a memory-intensive 7-thread program, where each thread sequentially copies the content from one private 64MB buffer to another buffer. We run an application for 20 seconds of simulated time, including 10 seconds of simulation of the warm-up process using KVM acceleration and 10 seconds of detailed simulation using the ARM O3CPU (about 20 billion cycles).

TABLE I: LC and BE Workloads

Latency-Critical (LC) Workloads	
Img-DNN (ID)	Image recognition
Moses (MS)	Real-Time translation
Xapian (XP)	Online search
Silo (SL)	In-memory transaction database
Masstree (MT)	Key-value store
Best-Effort (BE) Workloads	
Data Analytics (DA)	Bayes classification on a Wikimedia dataset
Graph Analytics (GA)	PageRank on a Twitter dataset
In-memory Analytics (IA)	Collaborative filtering on user-movie ratings
iBench	Massive streaming read and write

Fig. 12 reveals LC tasks' 95th percentile latency when running alone, with increasing request per second (RPS) on the simulator. We measured the baseline on a 4-core cluster with 8MB LLC. The QoS target is configured as the knee of the curve [17], marked with horizontal dashed lines. We define *max load* of an LC task as the maximum RPS under the QoS target.

TABLE II: Kunpeng-like Gem5 Configurations

Parameters	**Value**
ISA	Aarch64 (64-bit ARM)
Private L1I,L1D Caches	64kB, 4-way, 64B line size, 2 cycle hit, LRU, 4 mshrs
Private L2 Caches	512kB, 8-way, 64B line size, 12 cycle hit, LRU, 20 mshrs
Shared L3 Cache	2MB per CPU core, 16-way,64B line size, 32 cycle hit, LRU, 40 mshrs
Fetch/Decode/Issue/Commit	8 wide
ROB Entries	192
Issue/Load/Store Queue	64/32/32
Main Memory	1 channel, 16GB: DDR4-2400 x64, 8x8 Micron MT40A1G8
OS	Linux-5.10.137

TABLE III: ARM Neoverse-like Gem5 Configurations

Parameters	**Value**
ISA	Aarch64 (64-bit ARM)
Private L1I,L1D Caches	64kB, 4-way, 64B line size, 2 cycle hit, LRU, 16 mshrs
Private L2 Caches	512kB, 8-way, 64B line size, 8 cycle hit, LRU, 32 mshrs
Shared L3 Cache	2MB per CPU core, 16-way,64B line size, 10 cycle hit, LRU, 128 mshrs
Issue	14 wide
Fetch/Decode/Commit	8 wide
ROB Entries	316
Issue/Load/Store Queue	64/76/58
Main Memory	1 channel, 16GB: DDR4-2400 x64, 8x8 Micron MT40A1G8
OS	Linux-5.10.137

VI. Evaluation

A. Effect of Pivot

The comparison between Pivot and pure hardware approaches (i.e., MBA and MPAM) has been made in Sections II and III. This section compares Pivot against state-of-the-art work based on hardware-software co-design to demonstrate that, due to the limitations in the underlying hardware design (§II-B), even the co-design approach is insufficient. We compare Pivot with the following baselines: **Default**, which represents free contention for everything; **PARTIES** [17] and **CLITE**, that both utilizes Intel CAT [50] and MBA [8].

Due to the abundance of potential application mixes, we select a few representative scenarios. We tested three co-location scenarios: co-location of 1 LC task and BE tasks, co-location of multiple LC tasks and BE tasks, and co-location of multiple LC tasks.

1) Co-location of 1 LC task and BE tasks: We configure the load of 5 diverse LC tasks to 10%, 30%, 50%, 70%, and

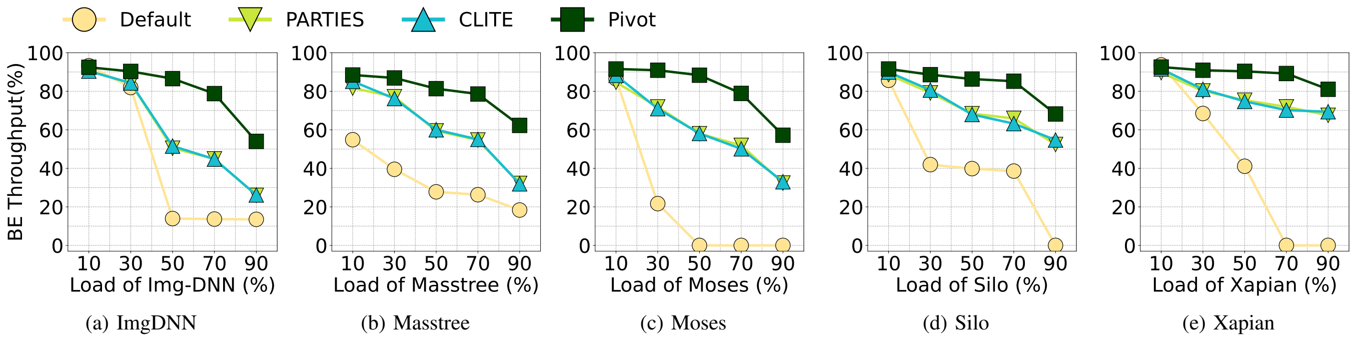


Fig. 13: Co-location of 1 LC and BE tasks (iBench). The Y-axis is the throughput (as a percentage of max throughput) of the BE task when the LC task is at a given RPS (X-axis) and meets QoS (higher is better).

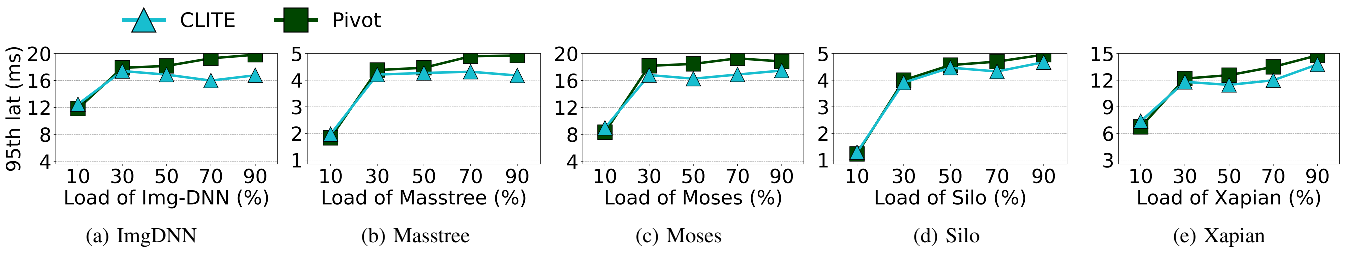


Fig. 14: The 95th percentile tail Latency of LC tasks in 1 LC and BE tasks co-locating scenarios.

90% of the *max load*, and adjust the number of threads for the BE task (iBench) from 0 to 7 at each specified load level on a cluster. Then, we evaluate the throughput of the BE task when the LC task meets QoS. The baseline (100%) throughput for the BE task is defined as the total IPC achieved by a 7-thread BE task executing independently in the simulator.

Fig. 13 illustrates the performance of this scenario. PIVOT outperforms all three policies as shown in Fig. 13. Concerning effective machine utilization (EMU) [45], which reflects the total load of all co-located tasks under QoS, *Default*, PARTIES, CLITE, and PIVOT achieve average 86.1%, 116.0%, 116.3%, and 133.2% EMU, respectively. Moreover, PIVOT achieves up to 2.06× higher BE throughput than thread-centric methods.

2) Co-location of 2 LC tasks and BE tasks: In this section, we utilize several representative 2 LC tasks co-located with BE scenarios to illustrate the effect of PIVOT.

Firstly, we present the result of the 2 LC apps co-located with iBench scenarios. We adjust the load of 2 LC tasks and the number of threads for the BE task (from 0 to 6). Then, we evaluate the throughput of BE tasks when 2 LC tasks meet QoS. The baseline (100%) is defined as the throughput achieved by a 6-thread BE task running independently.

Fig. 15 shows the effect of PIVOT. Unlike thread-centric resource managers, PIVOT can manage shared resources without introducing additional latency and extracting resources from non-critical load instructions. In co-location of Xapian and Img-DNN (Fig. 15d), PIVOT achieves on average 67.9%, 20.3%, and 15.7% higher EMU than *Default*, PARTIES and CLITE, respectively. In co-location of Moses and Img-DNN (Fig. 15h), PIVOT achieves on average 67.9%, 20.3%, and 15.7% higher EMU than *Default*, PARTIES and CLITE, respectively. Furthermore, Pivot enhances the BE task's throughput by up to 2.76× compared to CLITE.

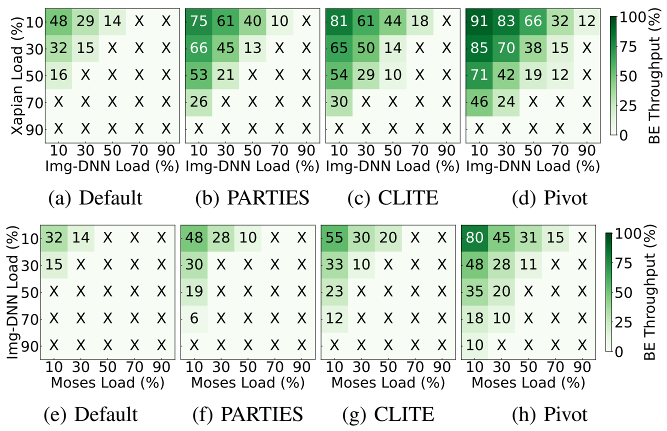


Fig. 15: Colocation of Xapian, Img-DNN, iBench and colocation of Moses, Img-DNN, iBench. Each cell represents the max throughput of BE tasks (higher is better) when LC tasks run at a given RPS, and both applications meet QoS.

To show the effect of Pivot in real-world applications, We present six co-location scenarios, including a single BE task and 2 BE tasks running with LC tasks.

Fig. 16 presents the single BE task's throughput when co-locating with 3 specific scenarios. Pivot acquire throughput improvement varying from 1.24× to 1.98× compared to CLITE. Fig. 17 presents the 2 BE applications co-location results. Pivot acquire throughput improvement varying from

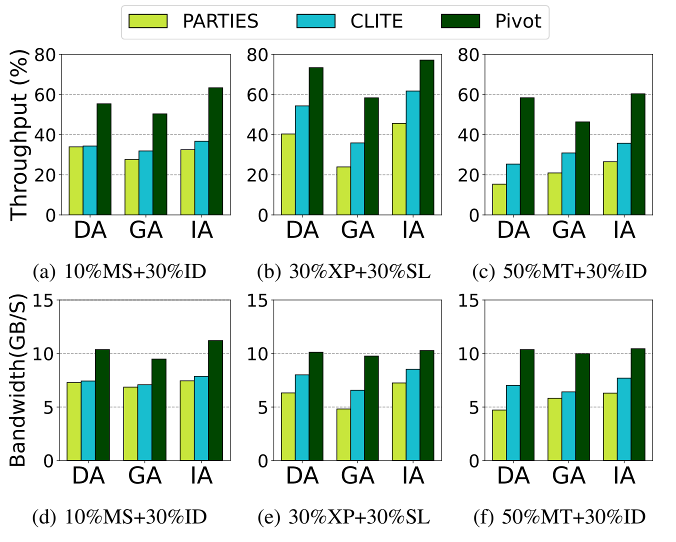


Fig. 16: Throughput of BE tasks (CloudSuite) and average memory bandwidth, when co-located with different indicated LC tasks mixes (higher is better).

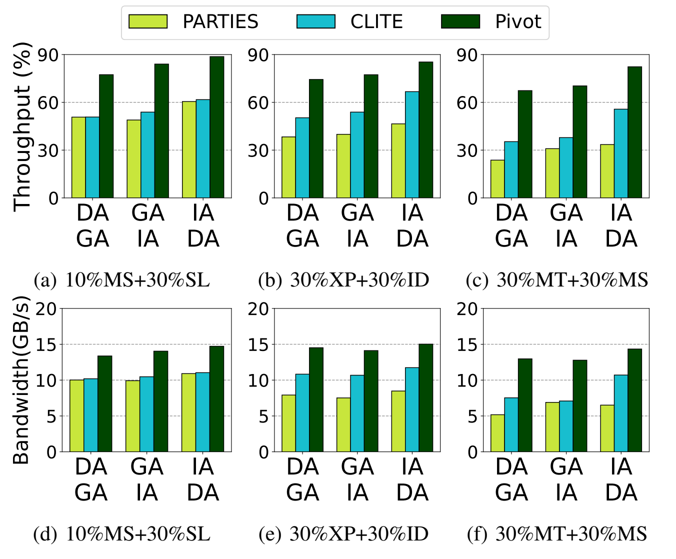


Fig. 17: Co-location of 2 LC and 2 BE tasks from CloudSuite. The Y-axis is the normalized throughput of the 2 BE tasks compared to running alone and average memory bandwidth when co-locating with different LC tasks (higher is better).

1.29× to 1.90× compared to CLITE. Since *Default* can not meet the QoS requirement, We do not present its result.

3) Co-location of Multiple LC tasks: We select 5 representative 2-app co-location scenarios and a 3-app co-location scenario on a cluster to demonstrate that PIVOT can also reduce memory source contention among LC tasks.

Given the diverse memory resource preferences among applications, the scenarios are determined by the complementary or similar preferences of the 2 tasks. In 2-app co-location scenarios, We configure the RPS of one LC task to 10%, 30%, 50%, 70%, and 90% of the *max load*, and sweep the load of another LC task from 5% to 100% of its respective *max load*, in 5% load increments. All load combinations that satisfy QoS requirements with each resource manager are documented. Fig. 18 presents the performance of 5 2-app co-located scenarios.

Default, which does not implement resource isolation, performs significantly worse than other resource managers. Compared to PARTIES, CLITE outperforms by simultaneously tweaking multiple resources to achieve the optimal configuration in some cases.

PIVOT assigns priority to control memory bandwidth without introducing additional latency. As depicted in Fig. 18e, Img-DNN and Moses are memory bandwidth-sensitive applications, PIVOT enhances the parallelizability of these applications by prioritizing critical load. Consequently, PIVOT achieves an average of 43.0%, 17.2%, and 12.1% higher EMU than *Default*, PARTIES, and CLITE in the experiment.

To show a more complicated case, We present the 3 LC tasks co-location result in Fig. 19. We test the four methods when Img-DNN is at low load and high load. PIVOT achieves up to 19.0% higher EMU than CLITE.

B. Evaluation of Load Criticality Prediction Method

This section discusses the effects of several load-criticality prediction methods to address memory bandwidth contention.

CBP [29] is a load criticality predictor near the ROB that guides the memory controller scheduling based on the ROB stall cycles. We use the BlockCount CBP to rank load requests on the memory controller. Since other MSCs, in addition to the memory controller, need to enforce priority, we also assess the performance of using the Binary-CBP prediction method to flag load without profiling (CBP + full path).

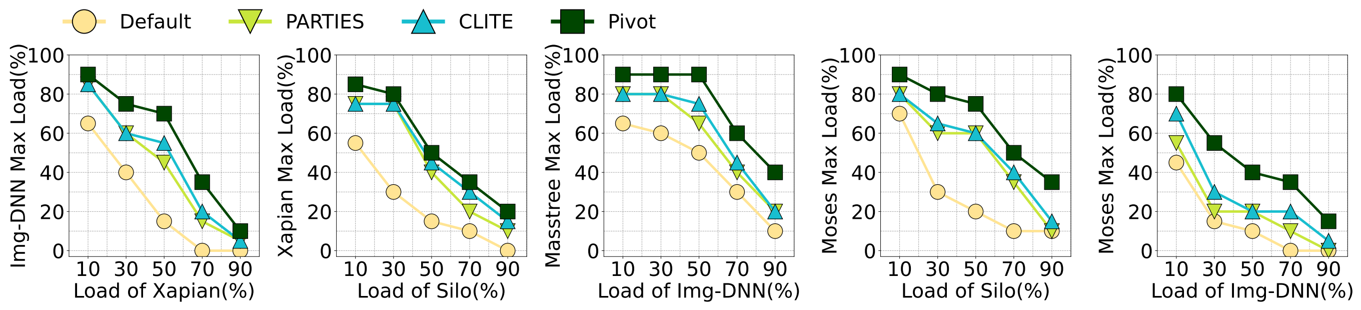
Fig. 20 illustrates the performance of four critical load prediction methods. CBP focuses on optimizing memory controller scheduling. Due to the absence of other MSCs isolation, LC tasks can only co-locate with fewer BE tasks' threads BE tasks to meet the QoS requirement. Although utilizing Binary-CBP to predict load criticality performs better, some memory access delays from some load instructions that cause ROB blocking are tolerable. Additionally, lacking profiling also leads to more misprediction due to the limited table.

Therefore, *CBP*, *Full Path + CBP*, achieve on average 39.0% and 12.0% EMU lower than PIVOT, respectively.

C. Sensitivity Analysis

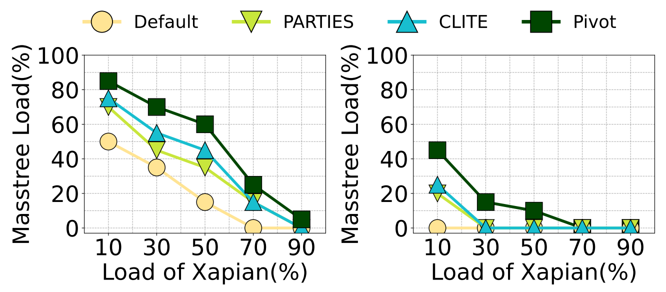
We use the five scenarios where one LC task at 70% of its *max load* co-located with the BE task (iBench) as a training set to conduct sensitivity analysis.

We evaluate four RRBP table sizes (16, 32, 64, and 128 entries) and compare them against a fully associative table featuring unlimited entries, which enables unaliased prediction, to explore the configuration of RRBP Table Size. As shown in Fig. 22, the RRBP table with 16 entries and 32 entries experiences up to 6.9% and 1.0% performance degradation by aliasing, respectively. Transitioning from an unlimited number of entries to a 64-entry predictor results in almost no



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

Fig. 18: Co-location of 2 LC tasks. The Y-axis is the maximum RPS (as a percentage of *max load*) of the second LC task (higher is better) when the first LC task is at a given RPS (x-axis) and both tasks meet QoS.



(a) 10% Img-DNN (b) 70% Img-DNN

Fig. 19: Co-location of Xapian, Masstree and, Img-DNN. Maximum RPS that Xapian (X-axis) and Masstree (Y-axis) can achieve (higher is better) when co-located with high (70%) and low (10%) load ImgDNNs.

performance drop. This results from profiling filtering out a subset of loads from the thousands of instructions. Moreover, almost no more than 64 of these loads from the LC tasks simultaneously reside in the RRBP Table.

We explore three interval lengths for the RRBP table refresh (500K, 1M, and 2M cycles). Compared to 500K and 2M refresh cycles, an RRBP table with a 1M cycle refresh period increases EMUs by averages of 0.5% and 1.3%, respectively.

We evaluated the configuration of LLC miss rates and the ROB stall rankings for offline profiling. When the configuration of miss rates was configured at 5% and 15%, the average EMU decreased by 0.9% and increased by 0.1%, respectively, compared to the current configuration in Section IV-B. When the configuration of stall ranking was configured at 10% and 15%, the EMU decreased by 0.6% and 1.8%, respectively.

D. The Effect of PIVOT on the ARM Neoverse CPU

This section evaluates the performance of PIVOT on the ARM Neoverse-like CPU model in Table III to more comprehensively validate our approach's effectiveness on modern server-class CPUs.

We compared PIVOT and CLITE using the same aforementioned experimental setup as in Fig. 13, Fig. 16, and Fig. 17.

Fig. 23 illustrates the performance of the scenarios where 1 LC and BE tasks are co-locating. PIVOT also outperforms CLITE in this model and achieves up to 2.11× higher BE throughput than CLITE.

To show the effect of PIVOT in real-world applications on the ARM Neoverse CPUs, We present the six same co-location scenarios. Fig. 24 presents the single BE task's throughput when co-locating with 3 specific scenarios. PIVOT acquire throughput improvement varying from 1.22× to 2.01× compared to CLITE. Fig. 25 presents the 2 BE applications co-location results. PIVOT acquire throughput improvement varying from 1.30× to 1.80× compared to CLITE.

VII. LIMITATION OF PIVOT AND FUTURE WORK

PIVOT have two potential limitations. Firstly, since PIVOT prioritizes scheduling critical loads and tolerates the latency of non-critical loads, PIVOT only achieves weak isolation. In contrast, MBA reduces the request rate of interfering tasks through additional latency, resulting in stronger isolation. Weak isolation harvests resources in the data center, but the irregular distribution of critical loads may cause PIVOT to slightly increase the average latency of LC tasks in some co-location scenarios. Strong isolation effectively enhances the average performance, even though it leads to resource under-utilization. Therefore, when LC tasks require lower average latency, future work should consider how to trade off the use of PIVOT and the strong isolation to meet the demands of actual scenarios.

Additionally, PIVOT needs offline profiling to filter loads. Hence, PIVOT cannot handle unknown LC tasks in multi-tenancy applications. Fortunatelly, the smaller instruction footprints in microservices and serverless scenarios provide an opportunity for PIVOT'S future work to apply in clouds.

VIII. RELATED WORK

The most relevant related work (e.g., MBA and MPAM) has been compared across the paper. This section presents the other related work.

A. Task Scheduling and Resource Management Mechanisms in the Data Centers

To improve resource utilization in data centers, earlier work [12], [21]–[23], [32], [47], [48] analyzes job resource

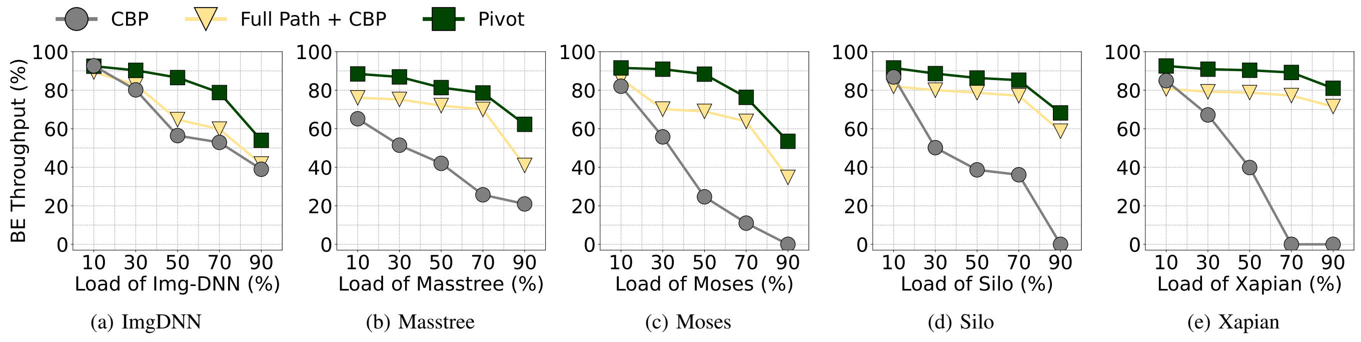


Fig. 20: Max BE tasks throughput using different critical load prediction methods in 1 LC and BE tasks co-locating scenarios (higher is better). The Y-axis is the throughput (as a percentage of max throughput) of the BE tasks when the LC task is at a given RPS (x-axis) and meets QoS.

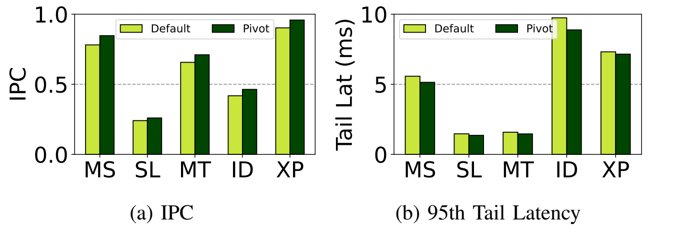


Fig. 21: The IPC and 95th tail latency of LC tasks at 70% *max load* running alone.

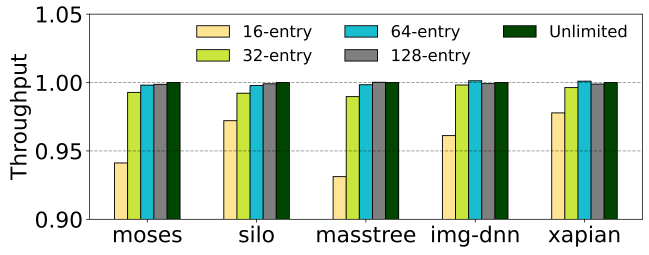


Fig. 22: Normalized BE tasks throughput reduction(with respect to the throughput of unlimited RRBP Table) while co-locating with diverse LC tasks at 70% *max load* (higher is better). All LC tasks meet QoS requirements.

sensitivities through offline profiling and collocates only those jobs that do not compete for the same resources to ensure QoS. Recent studies employ resource partition mechanisms. Some prior studies have explored partitioning a specific shared resource (e.g., LLC, memory bandwidth) in a data center environment. KPart [24], Vantage [58], SWAP [64], AET [31] and DCAPS [67] concentrate on optimizing cache partition techniques to mitigate resource contention within caches. MBP [33], MCP [49], ABP [68], and Hypart [55] explore memory bandwidth partition methods from various perspectives. Shenango [52] and Caladan [28] aim to achieve microsecond-scale resource partitioning, focusing solely on partitioning CPU cores. Stretch [46] explores the ROB partitioning method. However, these studies extensively examine the partitioning of one-dimensional resources but lack research on resource partitioning in other dimensions.

A significant portion of research is dedicated to multi-resource partitioning strategies in data centers. Specifically, some studies [15], [36], [45], [69]–[71] address scenarios where a single LC job is combined with multiple BE jobs. While these studies primarily target meeting the QoS demands of LC jobs, they overlook enhancing the execution efficiency of BE jobs. For instance, Heracles [45] is designed to meet the QoS requirements of a single LC job without implementing resource partitions for BE jobs, allowing them to run unmanaged. Therefore, Some work [16], [17], [44], [56] focus on co-locating multiple jobs using multi-resource partitioning. Pivot offers instruction-centric resource partitioning, enhancing data center resource utilization.

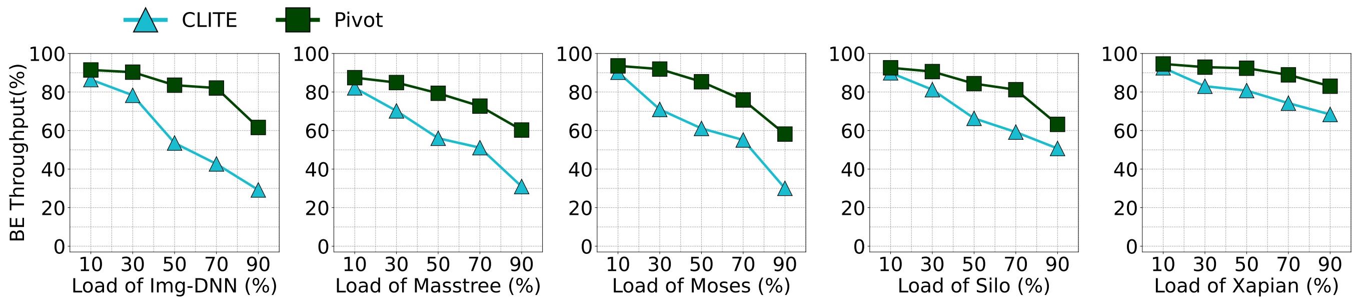
B. Critical Load Prediction

This section explains why the existing techniques on predicting and prioritizing loads are unsuitable for problem domain PIVOT targets (i.e., bandwidth partitioning in data centers). At a high level, with PIVOT, the critical loads for bandwidth partitioning cause a long stall on the reordering buffer. However, existing techniques are designed for different problem domains and thus are not ineffective in identifying such critical loads.

Das et al. [19] enhance application-level throughput by prioritizing packets belonging to stall-time critical applications over others in the network-on-chip. This work does not involve predicting load criticality and thus is orthogonal to PIVOT; The two techniques can be integrated for better effects.

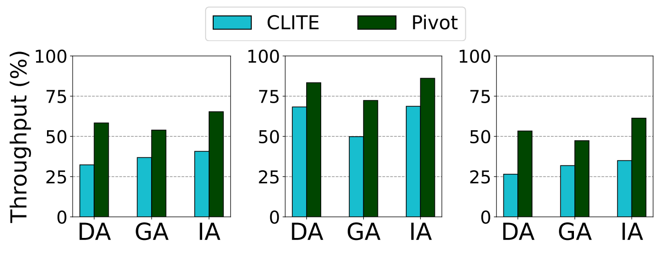
CLPT [60] utilizes the count of direct consumers to assess the criticality of a load instruction. However, the count of consumers is not an appropriate metric in bandwidth partitioning. This is because the criticality of the load in bandwidth partitioning (i.e., a load that causes a long stall in ROB) is not directly correlated with the number of consumers.

CATCH [51] targets optimizing cache prefetch and uses a construction of the data dependency graph (DDG) [27] to tag critical load on the costliest execution path. Similarly



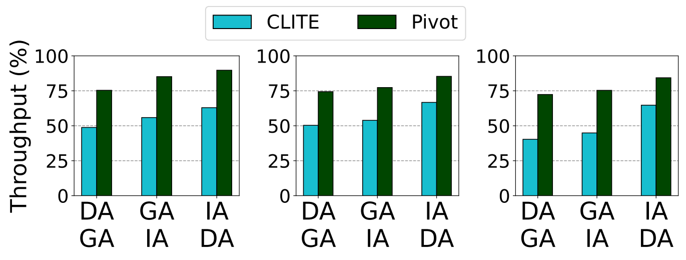
(a) ImgDNN (b) Masstree (c) Moses (d) Silo (e) Xapian

Fig. 23: Co-location of 1 LC and BE tasks (iBench) on the ARM Neoverse-like CPUs.The Y-axis is the throughput (as a percentage of max throughput) of the BE tasks when the LC task is at a given RPS (x-axis) and meets QoS (higher is better).



(a) 10%MS+30%ID (b) 30%XP+30%SL (c) 50%MT+30%ID

Fig. 24: Throughput of BE tasks (CloudSuite), when co-located with different indicated LC tasks mixes (higher is better).



(a) 10%MS+30%SL (b) 30%XP+30%ID (c) 30%MT+30%MS

Fig. 25: Co-location of 2 LC and 2 BE tasks from CloudSuite. The Y-axis is the normalized throughput of the 2 BE tasks compared to running alone and average memory bandwidth when co-locating with different LC tasks (higher is better).

to the discussion above, this metric is not well-suited for the bandwidth partitioning scenario since it is not directly correlated with a load that causes a long stall in ROB. CATCH also introduced complex hardware overhead.

CRISP [43] is a technique that predicts loads that frequently cause LLC misses and with low memory parallelism (MLP) as critical. CRISP is proposed to prefetch load slices and optimize the IPC of applications. Hence, CRISP does not distinguish criticality in memory access loads.

CBP [29] marks a load when it stalls the ROB in runtime without any profiling. The larger instruction footprint in the data center would limit the prediction accuracy of CBP. This is because more load instructions are hashed into the same table entry, resulting in CBP prediction failure.

Hermes [10] conceals the cache access time of long-latency load instructions via off-chip load prediction. However, Hermes does not distinguish the off-chip load, even if some do not affect performance and can be hidden in the OOO execution.

Some other research delves into investigating instruction criticality to enhance energy efficiency [7], [13], [41], [59]. However, these methods do not optimize the application's performance and do not apply to scenarios where QoS must be strictly guaranteed.

IX. Conclusion

This paper presents Pivot, a lightweight and general memory bandwidth partitioning mechanism that effectively resolves the inherent tension between enforcing the QoS requirements for LC tasks and maximizing bandwidth utilization, and thereby simultaneously achieving both of them. The effectiveness of Pivot comes from two key novel design insights that significantly depart from prior approaches. First, the priority of memory accesses issued by LC tasks should be enforced across *all* the components on the memory path. Second, with out-of-order execution, it is sufficient to prioritize only the scheduling of a selective portion of performance-critical loads issued by LC tasks. Pivot proposes a novel two-phase profiling technique that accurately identifies performance-critical loads in the LC task with minimal performance overhead, aligning with the data center deployment scenario. The memory access scheduling mechanism in Pivot prevents BE tasks from starvation and considers the scenarios where multiple LC tasks are co-located on the same physical node. Our extensive evaluation shows that Pivot improves effective machine utilization by up to 34.5% while increasing the throughput of the BE applications by up to 2.76×, without causing QoS violation.

Acknowledgment

We sincerely thank our shepherd and the anonymous reviewers for their insightful suggestions. The research is supported in part by the National Key R&D Program of China under Grant No. 2022YFB4500701, and by the National Science Foundation of China (Nos. 62032001, 62032008, 62332021, 62372011, 62472007). Diyu Zhou is the corresponding author.

X. Artifact Appendix

A. Abstract

We provide the source code and setup necessary for PIVOT. PIVOT is a hardware extension that provides instruction-centric memory bandwidth partitioning for data center applications.

PIVOT prioritizes critical load across the full memory access path and tolerates non-critical load's latency. We implemented ARM MPAM specification in the artifact and instantiated PIVOT based on MPAM. PIVOT has been implemented using gem5, a cycle-accurate CPU simulator.

This artifact consists of the simulator's source code, a disk image, a Linux bootloader, and a Linux kernel used for evaluation, as well as scripts needed to replicate the result in the paper.

B. Artifact check-list (meta-information)

- **Algorithm:** PIVOT.
- **Compilation:** GCC 10 or higher.
- **Binary:** All required binaries are included.
- **Run-time environment:** The simulator can be run on an ARM machine. We evaluated our artifact on the Huawei TaiShan 200 (Model 2280) with OpenEuler 21.09.
- **Hardware:** An ARM machine with more than 16 cores and at least 100 GB of disk space that supports KVM.
- **Execution:** Gem5 simulations.
- **Metrics:** Normalized IPC and 95th percentile latency.
- **Experiments:** Instructions to prepare evaluation resources, run the experiments, parse results, and plot graphs are available in the README file.
- **How much disk space required (approximately)?:** 100GB.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours.
- **How much time is needed to complete experiments (approximately)?:** About 30 hours on a 64 core system.
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** 10.5281/zenodo.14305038

C. Description

The artifact contains the source code of PIVOT along with resouces including a linux bootloader, a linux kernel and a disk images with benchmarks and datasets. This allows for reproducing the key experiment figure 3 contained in section §II-B, which comparing PIVOT to the modern bandwidth partitioning mechanisms Intel MBA and ARM MPAM (the most relevant related work).

1) How to access: The artifact can be downloaded from https://github.com/TELOS-syslab/Pivot or https://zenodo.org/doi/10.5281/zenodo.14305038. The resources for evaluation can be downloaded from https://zenodo.org/doi/10.5281/zenodo.14275908

2) Hardware dependencies: The artifact requires an ARM machine with more than 16 cores and at least 100GB of free disk space that supports KVM. We evaluated our artifact on the Huawei TaiShan 200 (Model 2280) with OpenEuler 21.09.

To avoid unknown issues caused by hardware inconsistencies, we can grant the reviewers access to our Huawei Taishan server for artifact evaluation.

3) Software dependencies: Gcc is used for compilation, Python3 to parse results, and Python3 with matplotlib to plot graphs. Additionally all the dependencies for Gem5 itself are needed. Detailed instructions on how to build can be found here: https://www.gem5.org/documentation/general_docs/building.

4) Data sets: The data sets for artifact evaluation is contianed in our preovided resource.

D. Installation

The scons software construction tool is used to compile the gem5 simulator. We provided a push button scripts to build workflow and download Pivot's resource.

```
git clone https://github.com/TELOS-syslab/Pivot
cd Pivot
./make.sh
```

E. Experiment workflow

The README provides detailed instructions required to reproduce the results from the paper. These include:

- First, download Pivot's resources and compile gem5 simulator on the platform.
- Then, execute the simulations (baseline, PIVOT, Intel MBA, ARM MPAM).
- Finally, parse the simulation results and plot the figure for normalized performance.

F. Evaluation and expected results

The expected results from this artifact is to recreate the key experiment results – Figure 3 (Comparing PIVOT against the most relevant related work Intel MBA and ARM MPAM).

G. Experiment customization

Scripts to conduct test are provided in the artifact *(folder scripts/*)*. Although customization is not expected, it can be done in interest of limited time or resources by changing a few parameters in the scripts in the artifact.

To run other benchmarks, users can utilized Qemu to modify the disk images.

References

[1] [Online]. Available: https://github.com/parsa-epfl/cloudsuite/

[2] [Online]. Available: https://redis.io/

[3] ARM architecture reference manual supplement memory system resource partitioning and monitoring (mpam) for ARMv8-A. [Online]. Available: https://developer.arm.com/documentation/ddi0598/latest

[4] ARM coresight performance monitoring unit architecture. [Online]. Available: https://developer.arm.com/documentation/ihi0091/latest

[5] Dynamic instrumentation tool platform. [Online]. Available: https://dynamorio.org/

[6] Akamai. Akamai online retail performance report: Milliseconds are critical. [Online]. Available: https://www.ir.akamai.com/news-releases/news-release-details/akamai-onlineretail-performance-report-milliseconds-are

[7] M. Alipour, S. Kaxiras, D. B. Schaffer, and R. Kumar, "Delay and bypass: Ready and criticality aware instruction scheduling in out-of-order processors," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 2020, pp. 424–434.

[8] H. Andrew, C. Marcel, and K. M. Abbasi. Introduction to memory bandwidth allocation. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-allocation

[9] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 2019, pp. 462–473.

[10] R. Bera, K. Kanellopoulos, S. Balachandran, D. Novo, A. Olgun, M. Sadrosadati, and O. Mutlu, "Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction," in *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*. IEEE, 2022, pp. 1–18.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011.

[12] S. Blagodurov, A. Fedorova, E. Vinnik, T. Dwyer, and F. Hermenier, "Multi-objective job placement in clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*. ACM, 2015, pp. 66:1–66:12.

[13] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*. ACM, 2015, pp. 272–284.

[14] D. Chen, D. X. Li, and T. Moseley, "Autofdo: automatic feedback-directed optimization for warehouse-scale applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*. ACM, 2016, pp. 12–23.

[15] Q. Chen, Z. Wang, J. Leng, C. Li, W. Zheng, and M. Guo, "Avalon: towards qos awareness and improved utilization through multi-resource management in datacenters," in *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*. ACM, 2019, pp. 272–283.

[16] R. Chen, H. Shi, Y. Li, X. Liu, and G. Wang, "Olpart: Online learning based resource partitioning for colocating multiple latency-critical jobs on commodity computers," in *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. ACM, 2023, pp. 347–364.

[17] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, p. 107–120.

[18] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 153–167.

[19] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Application-aware prioritization mechanisms for on-chip networks," in *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*. ACM, 2009, pp. 280–291.

[20] C. Delimitrou and C. Kozyrakis, "ibench: Quantifying interference for datacenter applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013, pp. 23–33.

[21] C. Delimitrou and C. Kozyrakis, "Qos-aware scheduling in heterogeneous datacenters with paragon," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, p. 12, 2013.

[22] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*. ACM, 2014, pp. 127–144.

[23] C. Delimitrou, D. Sánchez, and C. Kozyrakis, "Tarcil: reconciling scheduling speed and quality in large shared clusters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*. ACM, 2015, pp. 97–110.

[24] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 104–117.

[25] J. Evans, "Nvidia grace," in *2022 IEEE Hot Chips 34 Symposium, HCS 2022, Cupertino, CA, USA, August 21-23, 2022*. IEEE, 2022, pp. 1–20.

[26] M. Ferdman, A. Adileh, Y. O. Koçberber, 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 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*. ACM, 2012, pp. 37–48.

[27] B. A. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA 2001, Göteborg, Sweden, June 30-July 4, 2001*. ACM, 2001, pp. 74–85.

[28] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 281–297.

[29] S. Ghose, H. Lee, and J. F. Martínez, "Improving memory scheduling via processor-side load criticality information," in *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*. ACM, 2013, pp. 84–95.

[30] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: an analysis of alibaba datacenter traces," in *Proceedings of the International Symposium on Quality of Service, IWQoS 2019, Phoenix, AZ, USA, June 24-25, 2019*. ACM, 2019, pp. 39:1–39:10.

[31] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic modeling of data eviction in cache," in *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*. USENIX Association, 2016, pp. 351–364.

[32] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. V. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, 2009, pp. 261–276.

[33] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. B. Sullivan, I. Lee, and M. Erez, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*. IEEE Computer Society, 2012, pp. 53–64.

[34] B. Kasikci, W. Cui, X. Ge, and B. Niu, "Lazy diagnosis of in-production concurrency bugs," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 582–598.

[35] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: a technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. ACM, 2015, pp. 344–360.

[36] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sánchez, "Rubik: fast analytical power management for latency-critical systems," in *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. ACM, 2015, pp. 598–610.

[37] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[38] T. A. Khan, M. Ugur, K. Nathella, D. Sunwoo, H. Litz, D. A. Jiménez, and B. Kasikci, "Whisper: Profile-guided branch misprediction elimination for data center applications," in *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*. IEEE, 2022, pp. 19–34.

[39] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: Profile-guided instruction cache replacement for data center applications," in *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021*. IEEE, 2021, pp. 734–747.

[40] A. Kleen and B. Strong., "Intel processor trace on linux," in *Tracing Summit 2015 (2015)*.

[41] R. Kumar, M. Alipour, and D. Black-Schaffer, "Freeway: Maximizing MLP for slice-out-of-order execution," in *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 2019, pp. 558–569.

[42] R. Lavaee, J. Criswell, and C. Ding, "Codestitcher: inter-procedural basic block layout optimization," in *Proceedings of the 28th International Conference on Compiler Construction, CC 2019, Washington, DC, USA, February 16-17, 2019*. ACM, 2019, pp. 65–75.

[43] H. Litz, G. Ayers, and P. Ranganathan, "CRISP: critical slice prefetching," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 2022, pp. 300–313.

[44] Y. Liu, X. Deng, J. Zhou, M. Chen, and Y. Bao, "Ah-Q: quantifying and handling the interference within a datacenter from a system perspective," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE, 2023, pp. 471–484.

[45] 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, Portland, OR, USA, June 13-17, 2015*. ACM, 2015, pp. 450–462.

[46] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, "Stretch: Balancing qos and throughput for colocated server workloads on smt cores," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 15–27.

[47] 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 *44rd Annual IEEE/ACM International Symposium on Microarchitecture,MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*. ACM, 2011, pp. 248–259.

[48] A. Mirhosseini and T. F. Wenisch, "The queuing-first approach for tail management of interactive services," *IEEE Micro*, vol. 39, no. 4, pp. 55–64, 2019.

[49] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. T. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*. ACM, 2011, pp. 374–385.

[50] K. T. Nguyen. Introduction to cache allocation technology in the intel® xeon® processor e5 v4 family. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology/

[51] A. Nori, J. Gaur, S. Rai, S. Subramoney, and H. Wang, "Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*. IEEE Computer Society, 2018, pp. 96–109.

[52] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*. USENIX Association, 2019, pp. 361–378.

[53] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: A practical binary optimizer for data centers and beyond," in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 2019, pp. 2–14.

[54] M. Panchenko, R. Auler, L. Sakka, and G. Ottoni, "Lightning BOLT: powerful, fast, and scalable binary optimization," in *CC '21: 30th ACM SIGPLAN International Conference on Compiler Construction, Virtual Event, Republic of Korea, March 2-3, 2021*. ACM, 2021, pp. 119–130.

[55] J. Park, S. Park, M. Han, J. Hyun, and W. Baek, "Hypart: a hybrid technique for practical memory bandwidth partitioning on commodity servers," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018, Limassol, Cyprus, November 01-04, 2018*. ACM, 2018, pp. 5:1–5:14.

[56] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 193–206.

[57] A. Pellegrini, "Arm neoverse N2: arm's 2^{nd} generation high performance infrastructure cpus and system ips," in *IEEE Hot Chips 33 Symposium, HCS 2021, Palo Alto, CA, USA, August 22-24, 2021*. IEEE, 2021, pp. 1–27.

[58] D. Sánchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA*. ACM, 2011, pp. 57–68.

[59] A. Sembrant, T. E. Carlson, E. Hagersten, D. Black-Schaffer, A. Perais, A. Seznec, and P. Michaud, "Long term parking (LTP): criticality-aware resource allocation in OOO processors," in *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. ACM, 2015, pp. 334–346.

[60] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh, "Criticality-based optimizations for efficient load processing," in *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*. IEEE Computer Society, 2009, pp. 419–430.

[61] R. Sugumar, M. Shah, and R. Ramirez, "Marvell thunderx3: Next-generation arm-based server processor," *IEEE Micro*, vol. 41, no. 2, pp. 15–21, 2021.

[62] L. Tang, J. Mars, and M. L. Soffa, "Compiling for niceness: mitigating contention for qos in warehouse scale computers," in *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*. ACM, 2012, pp. 1–12.

[63] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "Reqos: reactive static/dynamic compilation for qos in warehouse scale computers," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*. ACM, 2013, pp. 89–100.

[64] X. Wang, S. Chen, J. Setter, and J. F. Martínez, "Swap: Effective fine-grain management of shared last-level caches with minimum hardware support," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 121–132.

[65] V. M. Weaver, "Advanced hardware profiling and sampling (pebs, ibs, etc.): Creating a new papi sampling interface," Tech. Rep., 2016.

[66] J. Xia, C. Cheng, X. Zhou, Y. Hu, and P. Chun, "Kunpeng 920: The first 7-nm chiplet-based 64-core ARM soc for cloud services," *IEEE Micro*, vol. 41, no. 5, pp. 67–75, 2021.

[67] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "DCAPS: dynamic cache allocation with partial sharing," in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. ACM, 2018, pp. 13:1–13:15.

[68] M. Xie, D. Tong, K. Huang, and X. Cheng, "Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning," in *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. IEEE Computer Society, 2014, pp. 344–355.

[69] W. Zhang, W. Cui, K. Fu, Q. Chen, D. E. Mawhirter, B. Wu, C. Li, and M. Guo, "Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters," in *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*. ACM, 2019, pp. 58–68.

[70] Y. Zhang, J. Chen, X. Jiang, Q. Liu, I. M. Steiner, A. J. Herdrich, K. Shu, R. Das, L. Cui, and L. Jiang, "Libra: Clearing the cloud through dynamic memory bandwidth management," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 815–826.

[71] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*. ACM, 2016, pp. 33–47.

[72] M. Zini, D. Casini, and A. Biondi, "Analyzing ARM's MPAM from the perspective of time predictability," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 168–182, 2023.