Memory Tiering in Python Virtual Machine

Yuze Li
Virginia Tech
Blacksburg, USA
lyuze@vt.edu

Shunyu Yao
Virginia Tech
Blacksburg, USA
shunyu@vt.edu

Jaiaid Mobin
Rochester Institute of Technology
Rochester, USA
jm5071@rit.edu

Tianyu Zhan
Virginia Tech
Blacksburg, USA
davidz0121@vt.edu

M. Mustafa Rafique
Rochester Institute of Technology
Rochester, USA
mrafique@cs.rit.edu

Dimitrios Nikolopoulos
Virginia Tech
Blacksburg, USA
dsn@vt.edu

Kirshanthan Sundararajah
Virginia Tech
Blacksburg, USA
kirshanthans@vt.edu

Ali R. Butt
Virginia Tech
Blacksburg, USA
butta@vt.edu

Abstract

Modern Python applications consume massive amounts of memory in data centers. Emerging memory technologies such as CXL have emerged as a pivotal interconnect for memory expansion. Prior efforts in memory tiering that relied on OS page or hardware counters information incurred notable overhead and lacked awareness of fine-grained object access patterns. Moreover, these tiering configurations cannot be tailored to individual Python applications, limiting their applicability in QoS-sensitive environments. In this paper, we introduce **M**emory **T**iering in **P**ython VM (MTP), an extension module built atop the popular CPython interpreter to support memory tiering in Python applications. MTP leverages reference count changes from garbage collection to infer object temperatures and reduces unnecessary migration overhead through a software-defined page temperature table. To the best of our knowledge, MTP is the first framework to offer portability, easy deployment, and per-application tiering customization for Python workloads.

CCS Concepts: • **Software and its engineering → Runtime environments**; **Garbage collection**.

Keywords: virtual machine, garbage collection, memory tiering

ACM Reference Format:
Yuze Li, Shunyu Yao, Jaiaid Mobin, Tianyu Zhan, M. Mustafa Rafique, Dimitrios Nikolopoulos, Kirshanthan Sundararajah, and Ali R. Butt. 2025. Memory Tiering in Python Virtual Machine. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3759548.3763372

VMIL '25, Singapore, Singapore

ACM ISBN 979-8-4007-2164-9/25/10
https://doi.org/10.1145/3759548.3763372

1 Introduction

Python dominates software programming communities, ranking first on TIOBE [8] and IEEE Spectrum [9] in 2025 for its simplicity and extensive libraries. However, Python data types tend to consume more memory than their native counterparts [7] due to Python's own managed runtime. For example, a typical integer consumes $24 + (n \times 4)$ bytes (where n is the number of digits in Python but only 4 bytes in C/C++). The extra memory is used to store the type information and reference counting, along with other metadata, to maintain run-time states. Together with the fact that Python employs a Garbage Collection (GC) that delays memory reclamation, it drastically increases the amount of memory consumed compared to native code [7, 27]. This excessive memory overhead is particularly problematic in cloud computing and high-performance computing environments, where memory costs scale with usage. Python's memory inefficiencies can lead to increased costs, degraded performance, and limited scalability [50].

In recent years, cloud data centers have adopted more heterogeneous memory hierarchies to meet the capacity and performance demands of modern applications. This creates a mismatch between Python's memory behavior and traditional tiering approaches. For example, Non-Uniform Memory Access (NUMA) [32] – one of the earliest examples of tiered memory – has been widely used in building scalable and cost-effective memory systems [6, 47]. The emergence of new memory techniques, such as Compute Express Link (CXL) [15, 35], further enables memory expansion and alleviates the memory wall problem through software/hardware scheduling and tiering at the OS/runtime level [21, 34]. Upper-tier memory typically offers lower latency and higher

bandwidth but less capacity, while lower-tier memory provides greater capacity at the cost of higher latency and lower bandwidth. When managing dynamic working set sizes in a tiered memory system, it is crucial to orchestrate data across the hierarchy carefully. However, effectively managing Python applications in tiered memory systems remains a significant challenge. Existing solutions either lack awareness of fine-grained Python object accesses or are incompatible with CPython's runtime, preventing Python workloads from taking full advantage of tiered memory hierarchy.

Modern memory tiering systems primarily rely on two dominant approaches, each with notable limitations. The first, OS-level management [29, 31, 36, 38, 41, 44, 54], has been widely adopted in production. These systems typically use page faults [29, 32, 38, 56] or access-bit sampling [33, 41, 45, 56] to infer memory access patterns. However, such techniques operate at the granularity of OS pages (4KB), whereas Python objects are much smaller (typically 24–200 bytes). As a result, multiple objects share the same page, leading to false sharing and inefficient placement decisions. Furthermore, these approaches often struggle to balance accuracy and overhead. Sampling-based systems may incur high or even unbounded overhead [33, 55, 56]. Moreover, cloud-based Python applications often have diverse priorities and QoS requirements, but OS-based tiering applies a *uniform configuration to all workloads*, with no awareness of which pages belong to latency-sensitive jobs and which do not [38]. As a result, it cannot tailor tiering policies to workload-specific QoS needs, leading to suboptimal placement decision.

The second type of tiered memory approach is tightly coupled with runtimes, which either define a new programming model offering self-defined APIs [16, 25, 46] in languages that support native execution (*e.g.*, C/C++), or leverage garbage collection in modern memory-managed languages to place program data [37, 52, 53, 58]. These approaches can track data access frequencies at the fine-grained object level. However, the former type only targets native executions programmed in C/C++ and requires non-trivial source code instrumentation (*i.e.*, non-transparent), while the second type only focuses on JVM-based languages. Consequently, Python programs that run in tiered memory have been left unexplored in this track.

In summary, existing solutions for tiered memory management are agnostic to the QoS of the workload or optimized for languages with native execution support (*e.g.*, C/C++) or JVM-based runtimes. However, Python applications pose unique challenges due to their dynamic nature, high object allocation rates, and reliance on automatic memory management. This creates a fundamental mismatch between Python's memory behavior and traditional OS-level page-based tiering approaches.

We analyze the performance of existing state-of-the-art memory tiering solutions and uncover that they fall short of achieving customizability in deployment and a balance between tracing accuracy and overhead (Sec. 3). To address these challenges, we present Efficient Memory Tiering (MTP) for memory-intensive Python programs. The key idea of MTP is that Python's built-in *Reference Counting (refcount)*, which tracks how many variables point to each object, can be leveraged to infer Python object access patterns.

Our contributions are as follows:

- *Lightweight sampling framework*: A CPython-integrated system that monitors object hotness with configurable overhead bounds (Sec. 5.1 & 5.2).
- *Reference count-based object tracking*: A novel approach that leverages CPython's built-in reference counting to infer object access patterns (Sec. 5.3).
- *Adaptive tiering without OS changes*: Introducing eagerness-awared strategy with software-defined page table to enable seamless data movement between memory tiers (Sec. 6).

Using real-world memory-intensive Python workloads in an emulated CXL-tiered environment, our evaluation shows that MTP outperforms existing solutions in most cases. Across 33 comparisons (11 workloads, each with three-tier configurations), MTP outperforms TPP[38] in 25 cases, AutoNUMA [4] in 30, and MEMTIS [33], with MTP yielding speedups of up to 29%. Unlike existing solutions that require OS modifications or application changes, MTP operates entirely within the Python VM through two simple APIs for enabling tracing and configuring QoS-based policies. To our knowledge, MTP is the first approach to integrate fine-grained memory tiering directly within an interpreted language runtime.

2 Python VM and Garbage Collector

Understanding MTP's approach requires familiarity with CPython's memory management, which provides the foundation for our object tracking and migration strategies.

CPython [1] is the standard Python interpreter, written in C and used by over 99% of Python applications [10]. It works by parsing Python statements and generating abstract syntax tree (AST), compiling the AST to bytecode, and executing the bytecode in the Python virtual machine. In CPython, all objects are represented by the PyObject structure, which contains metadata including type information and memory management fields. Each PyObject has a *refcount* that tracks how many references point to it in the heap. An object is freed when its refcount drops to zero. The key insight behind MTP is that reference count changes correlate with object access patterns. While refcounts don't directly measure access frequency, their fluctuations during execution provide valuable signals about object hotness (Sec. 5.3).

Beyond reference counting, CPython includes a cyclic garbage collector [43] that handles reference cycles—groups of objects that reference each other but are unreachable from the program. The cyclic GC maintains three generational lists that track container objects by age, with newer objects

collected more frequently than older ones. Reachable objects have their reference counts restored, while unreachable objects in cycles are deallocated to prevent memory leaks. MTP reuses this cyclic-GC infrastructure to efficiently discover all live objects in the system without additional traversal overhead (Sec. 5.1).

This combination of reference counting and cyclic collection provides MTP with both fine-grained access information and comprehensive object discovery capabilities.

3 Motivation

We identify two key limitations in existing tiered memory solutions that motivate MTP: (1) inability to adapt to workload-specific QoS requirements, (2) poor trade-offs between tracing accuracy and overhead, and (3) lack of tiering compatibility with Python VM. We present a summary of the comparison with prior work in Table 1. In the remainder of the section, we investigate these factors in detail.

3.1 Adapting to Workload-specific QoS Needs

Existing OS-based memory tiering systems [29, 31, 38, 41, 44, 54, 57] apply one-size-fits-all policies across all applications, hence they are unable to adapt to the diverse QoS requirements of individual workloads. For latency-sensitive workloads, the OS may be unaware of performance requirements, leading to unnecessary memory migration that increases access latency and overhead. Conversely, in long-running background tasks, the OS might misclassify the workload as hot, pinning the entire Resident Set Size (RSS) in fast-tier memory and limiting resources for other applications. These limitations arise from the OS's reliance on general heuristics, which do not account for workload-specific characteristics. For example, a machine learning training job may benefit from keeping model parameters in fast memory while allowing gradient buffers to reside in slower tiers, but OS-level systems cannot make such application-aware distinctions. Previous work like HeteroOS [28] and AutoNUMA [4] highlights the inefficiencies of OS-level policies that overlook application-specific QoS needs, leading to suboptimal memory placement.

While AIFM [46] requires *application developers* to manually instrument code and tune object structures, MTP empowers *cloud providers* to transparently enable tiering for any Python application through a simple API (listing 1), allowing them to adjust parameters to balance the trade-offs between tiering memory efficiency and tracing overhead.

Motivation 1. Memory tiering systems must provide configurable policies that can adapt to workload requirements rather than applying rigid, one-size-fits-all approaches.

3.2 Balancing Tracing Accuracy and Overhead

Current OS-based solutions face a fundamental dilemma: achieving accurate memory access tracking requires techniques that introduce prohibitive overhead, while approaches with low overhead sacrifice the fine-grained accuracy needed for effective tiering.

Accuracy. Page fault–based methods [4, 38, 56] and page table scanning [41, 45] scale poorly with memory footprint. DAMON [41] reduces overhead by grouping contiguous pages, assuming uniform access within regions. However, MEMTIS [33] demonstrates that such misgrouping leads to suboptimal data placement. PEBS-based approaches [33, 44, 45, 56] scale with access frequency but face a sampling dilemma: sparse sampling misses critical access patterns while aggressive sampling incurs high overhead. In contrast, MTP enables high-resolution *object-level* tracing.

Overhead. Achieving high accuracy through page table scanning can consume up to 10% of CPU resources due to frequent sampling [41]. PEBS-based methods suffer from fundamental hardware costs including microcode assists, cache pollution, and interrupt handling overhead [2]. Both lack dynamic tuning interfaces, reducing flexibility for cloud and HPC workloads. While MTP also incurs overhead, it offers a *straightforward software interface* for cloud providers to balance tracing accuracy and performance.

Motivation 2. Effective tiering requires fine-grained access tracking with controllable overhead—a balance that existing solutions fail to achieve, particularly for Python's object-oriented memory patterns.

3.3 Integrating Tiering into Python VM

Prior work has leveraged garbage collection in modern managed runtimes to place program data in tiered memory [3, 30, 37, 51, 52, 58] in tiered memory. These approaches can track data access frequencies at a fine-grained object level with acceptable overhead, but they primarily target JVM-based languages, which are incompatible with the Python VM. Unlike the JVM, Python's garbage collector does not perform object relocation. Any attempts to move objects therefore incur additional overhead. Moreover, Python lacks the read-/write barriers present in the JVM, making it infeasible to directly track PyObject accesses.

Motivation 3. Due to the design difference between Python VM and JVM, the absence of Python VM-specific object access tracing and migration scheme necessitates a completely new design.

4 MTP Design

MTP is designed to track PyObject accesses and migrate the corresponding pages between fast (local) and slow (remote) memory. MTP is a fully ***runtime-based***, ***transparent***, and ***fine-grained object tracing*** tiering system for Python applications with ***controlled overhead***. MTP is a complete

Table 1. Comparison of prior memory tiering systems that focus on different aspects. MTP targets Python applications, supports object tracking, is fully transparent to developers, customizable for cloud providers, and requires no offline profiling.

Solutions	Access tracking mechanism	Criteria for thresholding	Sub-page tracing	Offline analysis	Transparency	Customizability	Memory media
TPP [38]	Page fault	Static access count	✗	✗	✓	✗	DRAM/CXL
DAOS [41]	Access bit sampling	Static access count	✗	✗	✓	✗	DRAM/ZRAM
MEMTIS [33]	HW-based sampling	Memory access distribution	✓	✗	✓	✗	DRAM/NVM/CXL
AIFM [46]	Smart pointer interposition	CLOCK replacement	✓	✗	✗	✓	Local/far memory
TrackFM [48]	Smart pointer interposition	CLOCK replacement	✓	✓	✓	✗	Local/far memory
Write-Rationing [3]	GC interposition	GC-guided	✓	✗	✓	✗	DRAM/NVM
Panthera [51]	Static analysis + GC interposition	Static allocation + GC-guided	✓(coarse-grained)	✓	✓	✗	DRAM/NVM
MTP (ours)	Refcount-based inference	Memory access distribution + eagerness-driven	✓	✗	✓	✓	DRAM/CXL

user-space C extension module for CPython that is orthogonal to any existing OS-based memory tiering system.

4.1 Overview

Listing 1 describes the MTP API. Once imported, *MTP.start()* invokes the C extension module and initiates the marking phase (Sec. 5.1) when fast-tier memory falls below *mem_pressure*. The *mask* parameter indicates the memory node mask for demoting cold data, while *sample_interval* controls the interval for sampling phases (Sec. 5.2). Finally, *MTP.end()* halts tracing and releases metadata.

Listing 1. MTP Python Interface

```
import MTP
MTP.start(mask, mem_pressure, sample_interval)
# Python code you want to take effect
MTP.end()
```

MTP targets cloud providers running tiered-memory-backed services. In serverless platforms, it can be embedded into Python environments to manage instances with customizable tiering policies, remaining fully transparent to end users. Its lightweight API supports QoS-aware orchestration; for example, integrating MTP with Kubernetes Memory Manager [5] enables dynamic tiering adjustments for:

- **Latency-sensitive jobs:** set high *mem_pressure* and medium *sample_interval* to reduce remote access latency.
- **Long-running background tasks:** set low *mem_pressure* and high *sample_interval* to maintain acceptable performance without impacting high-priority workloads.

However, such customization is not possible with OS-based tiering approaches [38].

4.2 Challenges

To achieve object hotness tracing for migration in tiered memory for Python applications, MTP faces several challenges:

(I) **How to obtain object access information in the CPython runtime?** Object-level access information is tightly linked to programming language runtime implementation. Prior approaches focus on two domains: The first one is to provide a customized set of APIs based on C/C++ for programmers. Object temperatures can be obtained by overloading C++ smart pointer and the dereference operator (*i.e.*, `->`) and marking some bits as hotness indicators [24, 46]. However, this method does not apply to Python virtual machine, which defines object structures in C[1]. The second one focuses on JVM-based languages (*e.g.*, Java, Scala) that instruments read/write barriers [3, 52, 53]. CPython lacks such barriers. Instead, each object type defines its own set of APIs, *e.g.*, `PyList_SET_ITEM()` implements `list->ob_item[index]`. Manually instrumenting bookkeeping operations for each API is impractical.

To address this, MTP leverages **reference counting** [11], a garbage collection strategy in CPython. Note that unlike JVM, refcount does not directly indicate a PyObject is accessed, since it is simply pushing and popping operands from the stack. This leaves the object temperature tracking effort to MTP. Instead, MTP monitors the **changes** of refcount to infer object temperatures (Sec. 5.3).

(II) **How to mitigate the extra run-time costs during tracing?** Since CPython only marks container PyObjects (*e.g.*, lists, sets, tuples) rather than all live objects [43], MTP has to take care of it. We identified two approaches to mitigate the cost: The first is enabling the `Py_TRACE_REFS` macro to track all live PyObject references [18], though this adds significant CPU overhead (up to 60%) due to list manipulation during each PyObject lifecycle. The second approach is leveraging CPython's cyclic-GC module [43], which marks live objects by recursively traversing container PyObjects. However, this is a stop-the-world operation, temporarily blocking the application.

To mitigate this, we observe that **the set of live Python objects is related to how CPython cyclic-GC behaves**. One can use CPython GC hints to eliminate unnecessary tracing and thus, to reduce overhead (Sec. 5.1).

[1]Efforts to build a high-performance C++ version of Python are still in early stages [20].

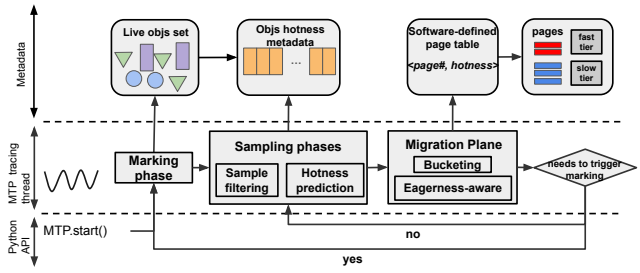

Figure 1. MTP Pipeline.

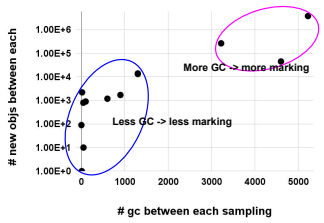

Figure 2. Marking phase frequency can be estimated by cyclic-GC frequency (x-axis) during run time.

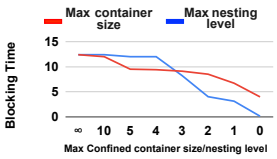

Figure 3. Application blocking time by setting different thresholds for container size and nesting level.

4.3 Pipeline

Figure 1 shows MTP's pipeline. Its principle is to periodically mark all live objects, observe refcount changes, and apply migration strategies accordingly. When enabled, MTP activates when local memory pressure is detected. First, in the marking phase (Sec. 5.1), all live objects are marked, generating PyObject temperature metadata. Next, MTP performs consecutive sampling phases (Sec. 5.2) to track PyObject refcount changes, filtering out unsuitable objects and calculating real-time hotness (Sec. 5.3).

After several sampling phases, MTP enters the migration plane to select migration candidates. It uses object-level hotness information to calculate the temperature of the corresponding memory pages (Sec. 6). The migration plane employs bucketing to classify hot and cold pages. MTP keeps hotness statistics up to date through periodic cooling and adaptively minimizes tracing overhead by monitoring CPython's cyclic-GC behavior during runtime. Note that all MTP's components run on a separate thread and do not interpose native CPython GC.

5 Object Access Tracing

This section presents the design to trace PyObject hotness in two key points: (1) marking and sampling phases to obtain and sample PyObjects' refcount changes, and (2) an online model to obtain real hotness values from those changes.

5.1 Marking Phase

To begin, MTP needs to know the scope of objects to sample. We noticed that CPython's native cyclic-GC module maintains three generational lists to detect cyclic references [43]. However, these lists only track containers PyObjects (*e.g.*, list, tuple, dict), meaning that non-container PyObjects (*e.g.*, an actual `PyLongObject` that is accessed) would be missed if MTP solely samples on them. To correctly mark all live PyObjects, MTP uses a built-in **recursive marking** approach, outlined in Algorithm 1. It starts by marking objects from the existing cyclic-GC lists (line 4). For PyObjects that are iterable, MTP leverages the `tp_traverse` method tailored to each object's type (line 11). If a child node is iterable, it recursively traverses its children (lines 13, 18), performing a depth-first search until no further child elements remain. Each marking phase appends newly initialized elements to the temperature metadata (lines 7 and 17).

MTP's marking has to be **stop-the-world**, meaning the application is temporarily paused. However, fewer marking phases can result in missing PyObjects created during run time. Therefore, MTP must balance marking overhead with accuracy. Fortunately, this can be inferred by observing CPython's cyclic garbage collection (GC) behavior. Figure 2 shows the relationship between the number of cyclic-GC events observed between two consecutive markings (x-axis) and the number of newly identified `PyObjects` in the same sampling period (y-axis). The trend reveals that when few cyclic-GC events occur (bottom-left region), the number of new objects is low, making a marking phase less worthwhile. Based on that, MTP temporarily skips current marking phase when cyclic-GC activity is less intensive. While this may slightly reduce marking accuracy, missed objects will be detected in subsequent markings. To remedy potential under-marking objects caused by consistently low cyclic-GC counts, MTP enforces one marking when it detects multiple consecutive moderate cyclic-GCs.

To further reduce overhead within a single marking phase, MTP applies two optimizations. First, it tracks both the *container size* and *nesting level* of the current container `PyObject` being traversed, and stops recursion immediately when either exceeds a preset threshold. Figure 3 shows how limiting these values controls the trade-off between marking accuracy and overhead. These two thresholds are dynamically tuned by observing last marking time: we set two thresholds unbounded when we observe short marking time. Conversely, we gradually constrain them (starting from the value of 10) when we observe longer marking time. Second, MTP maintains a `global_set` to record those already-traversed objects on the fly (lines 6, 16 in Algorithm 1), so that any PyObjects that are already in the set do not need to be recursively retraced. This is necessary since any PyObject can be referenced by one or multiple container PyObjects.

Algorithm 1: MTP Marking Phase.

```
Data: gc_list: global cyclic-GC list
      glob_set: global set for all live PyObjects
      hotness_arr: temperature metadata for refcnt
      changes
1  Function Do_Marking():
2      PyGILState_Ensure() // Acquire GIL
3      for node in gc_list do
4          obj = FROM_GC(node) // Obtain PyObject
5          if obj not in glob_set then
6              glob_set.insert(obj)
7              hotness_arr.append(obj) // Mark
8              Recursive_Visitor(obj)
9      PyGILState_Release() // Release GIL
10 Function Recursive_Visitor(obj):
11     traverse = Py_TYPE(obj).tp_traverse
12     if traverse then
13         traverse(obj, Traverse_Routine)
14 Function Traverse_Routine(inner_obj):
15     if inner_obj not in glob_set then
16         glob_set.insert(inner_obj)
17         hotness_arr.append(inner_obj) // Mark
18         Recursive_Visitor (inner_obj)
```

5.2 Sampling Phase

Having the scope of interested PyObjects during marking, MTP enters the sampling phase to sample their refcount changes. MTP leverages the system's endianness: it uses the unused 32 most significant bits (MSB) of the existing ob_refcnt field as a $growth$ counter, which **increases** regardless of whether the original refcount rises or falls.

During marking, MTP prepares the temperature metadata (lines 7 and 17 in algorithm 1). This metadata is stored as a global array, where each entry holds an 8-byte PyObject address ob, a 4-byte $prev\_growth$, and eight 1-byte fields to record refcount $changes$. MTP uses the following formula to sample changes: $changes[i]$ = ob.growth − prev_growth where $prev\_growth$ stores the previous $growth$ and is updated during each sampling. $changes[0...7]$ are later used to calculate hotness for that object.

Note that we choose to sample refcount instead of directly interposing refcount interfaces. While the latter approach seems viable, it incurs excessive CPU overhead, scaling up to 5 times, even if users do not intend to enable MTP during runtime. This is expected to be higher than interposing JVM's read/write barriers, which directly indicate object reads/writes, because the refcount in CPython occurs far more frequently than JVM barriers.

Sample filtering. Although sampling phases run asynchronously with the application thread, blindly sampling all objects could lead to unbounded duration — sampling 10 million PyObjects (which usually happens in modern workloads) takes around 1 second — delaying tracing recency. To mitigate this, we filter specific object samples under the following cases.

First, MTP skips two sampling phases for those PyObjects whose $changes[0...7]$ are all zeros (meaning they are not accessed). After that, all the skipped samples will be re-sampled in case they are accessed. Second, as any PyObject can be deallocated by CPython's main thread (GC) at any time and later on be replaced by another semantically different object at the same address. Accessing the $growth$ field of these temporarily "freed" objects will result in segmentation faults and crash the program. To address this, we add a signal handler for *SIGSEGV* in CPython's signal module and use *longjump* during sampling. If an invalid memory access occurs, MTP immediately marks those PyObjects as dropped to prevent further access.

Table 2. Different prediction models' performance on predicting PyObject real hotness based on refcount changes.

Models	MLR	MLR + RFE	Polynomial	Random Forest	MLP
Mean Squared Error	0.511	0.511	0.501	0.501	0.502
Mean Absolute Error	0.307	0.308	0.304	0.304	0.299
R^2	0.709	0.709	0.715	0.715	0.715
Pearson Correlation	0.842	0.842	0.846	0.846	0.846

5.3 Object Temperature Prediction

We conduct extensive *offline analysis* to correlate refcount changes with ground-truth temperatures from DAMON [41]. Empirical results show a typical pattern: *frequent but small refcount changes correlate with higher object temperatures*. While DAMON samples at coarse page or page-group granularity (Figure 4), we collect its data using the highest resolution and shortest interval, then align MTP's object addresses to DAMON's nearest page addresses to represent object-level hotness.

For each object-level refcount sample from MTP, we extract statistical features: median, standard deviation, count of non-zero refcount changes, and range. Then we use them as inputs to several prediction models (*e.g.*, MLP, MLR), with DAMON-provided temperatures as labels.

Table 2 summarizes model performance across various workloads (80/20 train-test split). All models predict PyObject temperature well. For example, MLR explains 70.9% of variance from the selected features (R^2 = 0.709) and achieves a Pearson correlation of 0.842, indicating a strong positive relationship with actual temperatures. Since complex models (e.g., MLP, Random Forest) yield only marginal gains with higher overhead, we adopt MLR in MTP.

Figure 4 illustrates the accuracy of the MLR model: the heatmap inferred from refcount changes (left, using 500ms *sample_interval*) closely aligns with the one generated from OS page table entry scans (right) for the same Python matrix multiplication workload. Although absolute temperatures

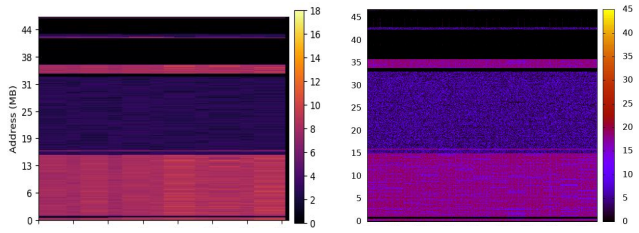

Figure 4. Heatmap generated from OS (right), and inferred from PyObject refcount changes (left).

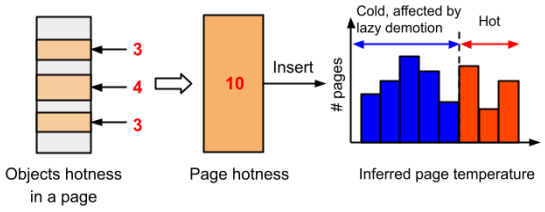

Figure 5. Software-defined page table contains objects hotness. We use bucketing to guide migration, while the hot/cold threshold is inferred by how eager hot pages are to be promoted.

may differ due to hotness range, the model can effectively distinguish hot and cold ones as long as their *relative* temperatures remain stable. This demonstrates that MTP can transparently capture object-level temperatures during run time *without OS page access information.*

6 Page Migration Strategy

After certain sampling phases, MTP triggers the migration plane. We choose to keep the migration unit at the page level due to Python VM constraints and the overhead of object-level migration (more details are discussed in Sec . 8).

First, we use a software-defined page temperature table that maps objects to their corresponding pages. Unlike the OS page table, our table is thinner, as it only tracks pages containing live PyObjects monitored by MTP. To decide the hotness of a page, MTP aggregates all objects' hotness inside that page. For instance, in Figure 5, a page with three PyObjects of hotness 3, 4, and 3 has a total hotness of 10. Next, MTP determines the threshold between hot and cold pages by applying bucketing and eagerness. After that, MTP demotes cold pages (if any) to the slow tier if there isn't enough space in the fast tier for hot pages. This frees up room to promote hot pages based on the available fast tier size.

MTP absorbs the page migration scheme from MEMTIS [33] but with an extension to improve migration efficiencies.

Existing design of page temperature cooling and access histogram-based bucketing in MEMTIS are incorporated into MTP, as explained below. With hotness stored in the page table, the temperature of a page is affected by periodic cooling. In other words, if the hotness of a page in the i^{th} interval is x, and the $(i-1)^{\text{th}}$ interval is $\text{hotness}_{(i-1)}$, the hotness in the i^{th} interval affected by cooling is as follows.

$$hotness_i = k \times x + (1-k) \times hotness_{i-1} \quad (1)$$

where k is a parameter to balance the weight of history temperature records and the current one.

Using hotness collected for page temperatures, MTP builds a histogram (Figure 5). In particular, the histogram consists of 10 buckets, and each bucket has a range of hotness following an exponential scale. For example, n^{th} bucket has the range of $\left[2^n, 2^{n+1}\right)$. The value of each bucket is the number of distinct base pages in the hotness range.

Determining hot/cold threshold for migration is challenging in MTP. In MEMTIS, a histogram is used to determine the hot page threshold, denoted as bucket index h. The total size of the pages in bin h and above is just below the fast memory capacity, and this threshold is periodically updated as the histogram changes. However, integrating such an OS-level scheme into MTP — or any user-level migration scheme — is challenging. MEMTIS has a global view of all system pages, making it straightforward to calculate the hot page threshold based on the fast tier capacity. In contrast, MTP only tracks the virtual pages of the current Python program, so using the fast tier capacity to set this threshold makes no sense. An alternative approach, similar to TPP, would involve pre-allocating a memory buffer to hold only newly identified hot pages. However, implementing this in CPython would require moving pages (objects) from the Python VM into this buffer, along with additional metadata to track updated addresses, which introduces both run-time and memory overhead.

New design. To accurately determine the hot/cold page threshold, the only solution is to demote some pages before promoting hot ones. MTP introduces an intelligent approach using a concept called *eagerness*, which quantifies how eager pages in the slow tier are to be promoted to the fast tier. *Eagerness* is a relative measure, compared to recent history, that reflects the changing extent to which pages in the slow tier are being accessed. It is computed using the following equation:

$$Eager_{\text{total[i]}} = RSS_{\text{fast}} * Eager_{\text{llc\_miss[i]}} * Eager_{\text{bw[i]}} \quad (2)$$

where i represents i's interval. RSS_{fast} is the percentage of pages currently located in the fast tier. $\text{Eager}_{\text{llc\_miss[i]}}$ is the z-score of pages encountering LLC load misses. It is computed as:

$$Eager_{llc\_miss[i]} = \frac{llc\_miss[i] - avg(prev)}{std\_dev(prev)} \quad (3)$$

where llc_miss[i] is the absolute LLC load misses ratio during interval *i*, and *prev* is a buffer holding recent LLC load misses. This equation measures how much the current LLC deviates from the expected range based on recent values. Similarly, $\text{Eager}_{\text{bw[i]}}$ is calculated by measuring memory bandwidth to the slow tier.

MTP uses this information to iterate through the histogram buckets. Starting from index zero and moving right, until it finds a bucket index *h* where the number of colder pages in the fast tier (needing demotion) just exceeds the number of hotter pages in the slow tier (needing promotion). This initial *h* is considered the most eager threshold. If $\text{Eager}_{\text{total[i]}}$ is smaller than $\text{Eager}_{\text{total[i-1]}}$, MTP decrements *h*, indicating that the current hot pages are less eager for promotion compared to the previous cycles.

7 Evaluation

To show the effectiveness of MTP, we evaluate it by answering the following questions:

- How does MTP perform with real-world Python workloads compared to state-of-the-art tiering solutions in an emulated CXL environment (Sec. 7.2)?
- What are the run-time and memory overhead for MTP (Sec. 7.3)?

7.1 Methodology

Hardware setup. In the absence of real CXL hardware, we emulate it on a dual-socket server, using one socket as the remote slow tier and the other as the local fast tier. Prior work [34] shows that CXL access latency is similar to cross-socket access latency. The fast-tier socket features a 16-core Intel(R) Xeon(R) Silver 4314 CPU at 2.40 GHz and various configurations of DRAM capacity. The slow-tier socket has 96GB DRAM capacity with all CPUs disabled. The CPU access latency to the slow tier is measured at 140ns, with a maximum bandwidth of 31 GB/s. We disable transparent huge pages, randomized virtual address space, hyper-threading, kernel same-page merging, Intel Turbo Boost, and memory swapping.

Workloads. We select 11 representative memory-intensive Python applications, including three SQL workloads that utilize SQLAlchemy [13] and eight graph computing workloads using the popular Python library NetworkX [26]. Table 3 shows detailed workload descriptions, including their memory consumption.

Comparison targets. We compare MTP against 3 systems: **TPP** [38], **AutoNUMA** [4], and **MEMTIS** [33]. We report the normalized performance compared to the baseline, where *each workload runs entirely in the remote slow tier (CXL)* without any tiering solution.

Table 3. Workloads specification and their memory usage.

Workloads	Descriptions	RSS
Astar	Returns a list of nodes in a shortest path between source and target using the A-star algorithm.	5.6G
Bellman	Computes shortest path lengths and predecessors on shortest paths in weighted graphs.	5.6G
BFS / BFS rand	Iterate over edges in a breadth-first-search starting at consecutive/random sources.	5.2G
Bidirectional	Returns a list of nodes in a shortest path between source and target using bidirectional search.	4.6G
KC	Returns the maximal subgraph that contains nodes of degree k.	8.6G
LC	Find the best partition of a graph using the Louvain Community Detection Algorithm.	3.9G
SP	Find shortest paths between nodes.	5.6GB
SQL1	Populate relational databases with different table structures and insertion and updates on random values using foreign key.	3.6G
SQL2		8.8G
SQL3		3.7G

Tiering Configurations. We configure fast tier size to 25%, 50%, and 75% of each workload's RSS (Table 3) to simulate different levels of fast-tier scarcity. To do that, we first modify the Linux kernel boot argument using the *memmap* GRUB option [42] to reserve most of the fast-tier memory. We then use a memory hogger to consume the remaining fast-tier memory until the target size is reached.

For MTP, we set *mem_pressure* to 1GB and *sample_interval* to 250ms. For TPP, we adjust the *demote_scale_factor* to 2% to trigger fast-tier memory reclamation. For MEMTIS, which relies on transparent huge pages (THP), we present its performance with THP enabled. For the other three solutions, we compare performance with THP disabled, as all these migration designs are based on base pages.

7.2 Performance Comparison

Figure 6 shows the normalized performance of all tiering solutions relative to running workloads entirely on slow tier. Across 33 configurations (11 workloads × 3 memory ratios), MTP outperforms TPP in 25 cases and AutoNUMA in 30. While MEMTIS outperforms MTP in 15 instances, MTP still achieves a competitive geometric mean performance (1.18 vs. 1.20).

Astar (Figure 6a) benefits from MTP 's MLR-based hotness inference via object refcounts, yielding 21% speedup over MEMTIS and up to 29% over AutoNUMA. TPP suffers from excessive kernel overhead — 53% runtime spent in kernel — due to aggressive migrations triggered by uniform page hotness, causing "ping-pong" effects. Similar issues occur in LC and SQL1 (Figures 6g, 6i). For MEMTIS, performance improvements compared to all slow tiers are marginal, with speedup ranging from 6% to 11% as the fast-tier size increases from 25% to 75%. Comparatively, MTP achieves the highest performance while maintaining stable run-time variance across fast-tier sizes.

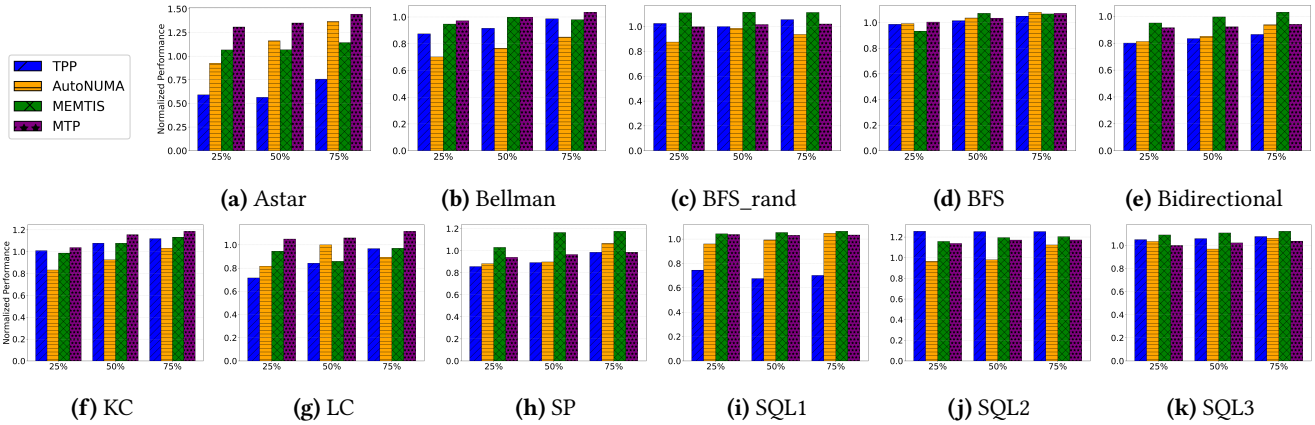

Figure 6. Performance of MTP and OS page-based tiering solutions under three tiering configurations (with the fast tier set to 25%, 50%, and 75% of total RSS), normalized to the performance of running the entire benchmarks in slow tier (CXL). Only MEMTIS is evaluated against the THP-enabled baseline. Across 33 comparisons, MTP outperforms TPP in 25 cases, AutoNUMA in 30, and MEMTIS in 15 cases.

In Bellman (Figure 6b), TPP has better performance compared to Astar, as it better distinguishes hot/cold pages. AutoNUMA performs the worst due to minor page faults. Similar patterns are observed in BFS_rand, KC, and SQL2. MTP still achieves the best result among all solutions.

In contrast, for BFS_rand, Bidirectional, and SP (Figures 6c, 6e, 6h), MTP 's initial marking cost dominates — especially there are massive amount of objects in Python VM — leading to high overhead during the first marking. Despite this, once MTP has collected the object information, it effectively distributes memory objects accordingly, as evidenced by the minor performance variations across different settings.

SQL workloads (Figure 6i, 6j, 6k) populate relational databases with different table structures and perform update/insertion on random values, with different scales and compute intensity. In this case, MTP performs better than AutoNUMA by an overall 5%. However, it is slightly slower than TPP by 5.1% and MEMTIS by 4.9% in SQL2 and SQL3, respectively.

Finally, comparing TPP and AutoNUMA, TPP achieves an 11.2% geometric mean advantage over AutoNUMA in memory-intensive workloads (Bellman, BFS_rand, KC, SQL2, and SQL3) by sampling slow-tier page faults and relying on LRU aging in the fast tier. This minimizes the overhead of temperature detection.

7.3 MTP Overhead Analysis

Run-time overhead: MTP 's overhead is dominated by its marking phase, accounting for over 95% of total overhead (Figure 7). Marking can add up to 8.3% (SP) to total run time, but this is mitigated by skipping unnecessary phases based on GC frequency. Most overhead arises from the initial marking phase, which is crucial for identifying the initial scope of PyObjects. This shows that, even under the context of unavoidable overhead, MTP remains competitive in overall performance compared to prior OS-based tiering approaches (Figure 6), demonstrating its effectiveness in classifying hot and cold objects for migration.

The ratio of marking time increases when the fast-tier size grows, as the absolute marking time remains stable while the total run time decreases, thereby raising this percentage. However, note that such growth is bounded, since MTP triggers marking only when fast-tier capacity is insufficient.

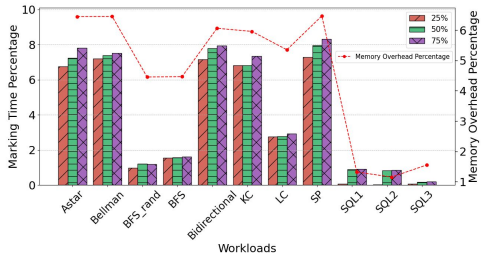

Figure 7. Marking time and memory overhead percentage for MTP. Marking time consists of at most 8.3% of total running time, while memory overhead consists of at most 6.4% of total RSS.

Memory overhead: In Figure 7 (dashed line), we show memory overhead as a percentage of RSS compared to vanilla CPython 3.12. This overhead, primarily from tracking PyObject temperature metadata, correlates with run-time overhead (bars), as both scale with the number of sampled objects. Memory overhead ranges from 1% to 6.4%, with graph workloads incurring more due to higher object creation than SQL workloads. Because Python treats everything as objects, MTP 's metadata tracking incurs more memory overhead than page-level OS approaches. One mitigation is storing metadata in the slow tier, though frequent access may delay

migration decisions. Alternatively, using a more compact metadata structure could reclaim some space.

Overhead discussion: Notably, MTP incurs **no** runtime overhead when it is not explicitly activated by the application (Listing 1), and likewise imposes no performance penalty under sufficient fast memory. In terms of memory footprint, MTP stores the *growth* field within unused bits of *ob_refcnt*, eliminating any additional memory cost even when tracing is disabled.

8 Limitations and Discussion

Inability to track native executions. Even though we put significant effort into realizing MTP and minimizing overhead, MTP can only track PyObjects with refcount changes in the Python VM. Hence, it misses native objects from libraries like NumPy, SciKit-Learn, or TensorFlow [7, 17, 39, 49]. A potential solution is detecting native execution and delegating migration to OS-level methods.

MTP vs. MEMTIS. While MEMTIS outperforms in certain scenarios, it operates at a coarse-grained, system-wide level, limiting its adaptability to workload-specific tuning. In contrast, MTP is a user-level solution that offers fine-grained control at both the application and code-segment level—without requiring any kernel modifications.

Importantly, MTP and MEMTIS are complementary and can coexist. For instance, they can run in separate VMs on the same server: MEMTIS can handle general-purpose tiering, while MTP optimizes Python workloads with specific QoS requirements.

Why MTP uses page-level migration. We do not incorporate object-level migration for two principal reasons. First, existing object-level approaches such as Write-rationing [3], Panthera [51] are tailored for JVMs whose garbage collector is *moving GC*. Thus, it's intuitive to bind GC-managed regions to different memory tiers. Object moving in virtual addresses automatically translates to movement among physical memory devices. However, forcibly adapting moving GC to Python VM violates its core GC and memory management design, making it impractical.

Second, even if this challenge is solved with engineering effort, forcibly enabling object-level migration may still cause more overhead than page migrations. Prior work in SemSwap [14] reveals that consolidating hot objects into dense pages could minimize network traffic. However, it introduces considerable overhead in metadata management and run-time address translation costs. Furthermore, DiLOS [59] shows that page migration imposes comparatively only little additional cost relative to object-level migration (~ 25% extra overhead for a 4KB page vs. a 128-byte object under one-sided RDMA reads). With CXL's low latency (~ 210ns), page-level migration is both practical and efficient, especially when hot objects are densely colocated.

MTP in GIL-free build. We acknowledge the no-GIL initiative (PEP 703) and Python 3.13's free-threaded build [19, 23], which enable true multi-threading. *Yet GIL removal primarily addresses bytecode parallelism, not global coordination.* Stop-the-world phases such as cyclic GC still remain [22]. As with cyclic GC, MTP must scan nearly all live `PyObjects` during marking, which cannot proceed safely alongside concurrent mutations. Therefore, these pauses arise from coordination requirements rather than the GIL itself, and a no-GIL interpreter does not remove MTP 's marking overhead.

GC non-intrusive design. JVM-based tiering systems such as Memliner [53] achieve fine-grained data placement by heavily modifying the native JVM garbage collector. This requires invasive changes to GC algorithms, creating substantial engineering complexity and tight coupling between tiering logic and the runtime. Such coupling increases maintenance burden when upstream GC evolves.

In contrast, MTP adopts a lightweight, non-intrusive strategy: it **reads** CPython's cyclic-GC lists for liveness markings and **samples** PyObject reference count changes to infer hotness, without altering the GC's decision-making. This decoupled design preserves CPython's memory management, reduces the risk of GC regressions, and allows integration with minimal runtime disruption. As a result, MTP avoids heavy CI/CD dependencies on native VM/GC logic and eases adoption across Python environments.

MTP atop GraalPython. GraalPython [40] is a Python 3 implementation on GraalVM [12], a runtime that supports languages like Java. It inherits JVM features such as moving GC, a generational heap, and built-in read/write barriers—enabling fine-grained hotness tracking and object-level decisions, as in prior JVM-based tiering systems [30, 51, 53].

9 Conclusion

In this work, we present MTP, a runtime extension built on CPython that enables transparent and efficient memory tiering for Python applications. It infers object hotness through refcount changes using a simple inference model and employs a software-defined page table with page bucketing for efficient migration. Experimental results show that MTP delivers comparable performance compared to state-of-the-art OS-based tiering solutions. Most importantly, MTP offers opportunities to customize tiering configurations based on each workload's QoS requirement that OS-based systems cannot achieve.

Acknowledgements

We thank our anonymous reviewers for their detailed feedback and valuable suggestions. This work is sponsored in part by the NSF under the grants: CSR-2106634 and CSR-2312785.

References

[1] 2025. CPython. https://en.wikipedia.org/wiki/CPython. Accessed: 2025-08-14.

[2] Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative evaluation of intel pebs overhead for online system-noise analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*. 1–8.

[3] Shoaib Akram, Jennifer B Sartor, Kathryn S McKinley, and Lieven Eeckhout. 2018. Write-rationing garbage collection for hybrid memories. *ACM SIGPLAN Notices* 53, 4 (2018), 62–77.

[4] Andrea Arcangeli. 2012. *AutoNUMA AutoNUMA Red Hat, Inc.* https://mirrors.edge.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf

[5] Kubernetes Authors. [n. d.]. Utilizing the NUMA-aware Memory Manager. https://kubernetes.io/docs/tasks/administer-cluster/memory-manager/

[6] Jeff Barr. 2022. New General Purpose, Compute Optimized, and Memory-Optimized Amazon EC2 Instances with Higher Packet-Processing Performance | AWS News Blog. https://aws.amazon.com/blogs/aws/new-general-purpose-compute-optimized-and-memory-optimized-amazon-ec2-instances-with-higher-packet-processing-performance/

[7] Emery D Berger, Sam Stern, and Juan Altmayer Pizzorno. 2023. Triangulating Python Performance Issues with {SCALENE}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 51–64.

[8] TIOBE Software B.V. 2025. TIOBE Index - TIOBE. https://www.tiobe.com/tiobe-index/. Accessed: August 22, 2025.

[9] Stephen Cass. 2025. The Top Programming Languages 2025 - IEEE Spectrum. https://spectrum.ieee.org/top-programming-languages-2024. Accessed: August 22, 2025.

[10] Bite Code. 2023. What's the deal with CPython, Pypy, MicroPython, Jython...? https://www.bitecode.dev/p/whats-the-deal-with-cpython-pypy?utm_source=chatgpt.com

[11] George E Collins. 1960. A method for overlapping and erasure of lists. *Commun. ACM* 3, 12 (1960), 655–657.

[12] Wikipedia Contributors. 2025. GraalVM. https://en.wikipedia.org/wiki/GraalVM

[13] Rick Copeland. 2008. *Essential sqlalchemy*. " O'Reilly Media, Inc.".

[14] Siwei Cui, Liuyi Jin, Khanh Nguyen, and Chenxi Wang. 2022. SemSwap: Semantics-aware swapping in memory disaggregated datacenters. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*. 9–17.

[15] Compute Express Link (CXL). [n. d.]. https://www.computeexpresslink.org/. https://www.computeexpresslink.org/

[16] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[17] Pedregosa Fabian. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research 12* (2011), 2825.

[18] Python Software Foundation. [n. d.]. 3. Configure Python. https://docs.python.org/3/using/configure.html#cmdoption-with-trace-refs

[19] Python Software Foundation. 2024. Python experimental support for free threading. https://docs.python.org/3/howto/free-threading-python.html

[20] Gil. 2023. Python C++ (EXPERIMENTAL + IN PROGRESS). https://github.com/gf712/python-cpp.

[21] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[22] Sam Gross. 2023. Correct C API Usage Logic for NO_GIL Multithreading (Issue #133). https://github.com/colesbury/nogil/issues/133. GitHub issue.

[23] Sam Gross. 2023. PEP 703 – Making the Global Interpreter Lock Optional in CPython | peps.python.org. https://peps.python.org/pep-0703/

[24] Zhiyuan Guo, Zijian He, and Yiying Zhang. 2023. Mira: A Program-Behavior-Guided Far Memory System. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 692–708. doi:10.1145/3600006.3613157

[25] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 417–433.

[26] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[27] Matthew Hertz and Emery D Berger. 2005. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 313–326.

[28] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 521–534.

[29] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for {Multi-Tiered} Memory Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 715–728.

[30] Iacovos G Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S Zakkak, Polyvios Pratikakis, and Angelos Bilas. 2023. Teraheap: Reducing memory pressure in managed big data frameworks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 694–709.

[31] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 317–330. doi:10.1145/3297858.3304053

[32] Christoph Lameter. 2013. An overview of non-uniform memory access. *Commun. ACM* 56, 9 (2013), 59–54.

[33] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 17–34.

[34] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[35] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S Berger, Marie Nguyen, Xun Jian, Sam H Noh, and Huaicheng Li. 2025. Systematic cxl memory characterization and performance analysis at scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1203–1217.

[36] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered Memory Management Beyond Hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 731–747.

[37] Wenjie Liu, Shoaib Akram, Jennifer B Sartor, and Lieven Eeckhout. 2021. Reliability-aware garbage collection for hybrid HBM-DRAM memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 1 (2021), 1–25.

[38] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 742–755.

[39] NumPy. 2022. NumPy documentation. https://numpy.org/doc/stable/.

[40] oracle. 2025. GitHub - oracle/graalpython: GraalPy – A high-performance embeddable Python 3 runtime for Java. https://github.com/oracle/graalpython

[41] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. 2022. DAOS: Data access-aware operating system. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 4–15.

[42] pmem.io. 2019. Using the memmap Kernel Option | Persistent Memory Documentation. https://docs.pmem.io/persistent-memory/getting-started-guide/creating-development-environments/linux-environments/linux-memmap

[43] python. 2017. Garbage collector design. https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md

[44] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 392–407.

[45] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 803–817.

[46] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[47] Amazon Web Service. 2024. Overview of performance and optimization options - Amazon EC2 Overview and Networking Introduction for Telecom Companies. https://docs.aws.amazon.com/whitepapers/latest/ec2-networking-for-telecom/overview-of-performance-optimization-options.html

[48] Brian R Tauro, Brian Suchy, Simone Campanoni, Peter Dinda, and Kyle C Hale. 2024. TrackFM: Far-out compiler support for a far memory world. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 401–419.

[49] TensorFlow. 2019. TensorFlow. https://www.tensorflow.org/.

[50] Pragati Verma. 2025. Understanding Python's Garbage Collection and Memory Optimization. https://dev.to/pragativerma18/understanding-pythons-garbage-collection-and-memory-optimization-4mi2?utm_source=chatgpt.com

[51] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 347–362.

[52] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A {Memory-Disaggregated} Managed Runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 261–280.

[53] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. {MemLiner}: Lining up Tracing and Application for a {Far-Memory-Friendly} Runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 35–53.

[54] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. doi:10.1145/3503222.3507731

[55] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad:{Non-Exclusive} Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 19–35.

[56] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. 2024. {FlexMem}: Adaptive Page Profiling and Migration for Tiered Memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 817–833.

[57] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 331–345.

[58] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020. Improving program locality in the GC using hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 301–313.

[59] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. Dilos: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 266–282.

Received 2025-07-21; accepted 2025-08-11