Efficient Memory Tiering for Python Applications

ABSTRACT

Modern Python applications, such as computing over graphs and database querying, consume massive amounts of memory in data centers. Emerging tiered memory technologies such as CXL offer promising solutions to mitigate this challenge by providing multiple memory layers with varying capacity, latency, and cost, aiming to balance efficiency with high memory demands. Previous approaches to memory tiering using OS-level information overlooked fine-grained object access patterns and introduced significant overhead. Moreover, these tiering configurations cannot be tailored to individual Python applications. Conversely, fine-grained temperature tracing at the object level is challenging, as it typically requires extensive application modifications, thus limiting its deployment scope. In this paper, we introduce Efficient Memory Tiering (EMT), an extension module built on top of the widely used CPython interpreter to enable memory tiering for Python applications. EMT leverages reference count changes from garbage collection to infer object temperatures and reduces unnecessary migration overhead through a software-defined page temperature table with adaptive lazy demotion. To the best of our knowledge, EMT is the first framework to offer portability, easy deployment, and opportunity to customize tiering configurations for Python applications.

1 INTRODUCTION

Python is one of the most popular programming languages for its simplicity and abundance of libraries; *e.g.*, it ranks first on TIOBE [7] and IEEE Spectrum [8] in 2024. However, Python data types tend to consume more memory than their native counterparts [6] due to its own managed runtime. For example, a typical integer consumes $24 + (n * 4)$ 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 [6, 25]. Python's ecosystem also includes numerous libraries covering a wide range of domains: graph computations [12, 24], database query [11, 38], machine learning [45, 52], web development kits [15, 41], and more. While this extensive collection of libraries makes Python a versatile and powerful language for developers and researchers alike, it also leads to significant memory usage. This excessive memory overhead is particularly problematic in cloud computing and high-performance computing environments, where memory costs scale with usage. In modern workloads such as machine learning and data analytics, Python's memory inefficiencies can lead to increased costs, degraded performance, and limited scalability.

In recent years, cloud data centers have adopted more heterogeneous memory hierarchies to meet the capacity and performance demands of modern applications. 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 [5, 50]. The emergence of new memory techniques such as Compute Express Link (CXL) [14] further allow memory expansion and alleviate the memory wall problem via software/hardware schedule and tiering at 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 essential to carefully orchestrate data across the hierarchy.

There are two dominant approaches to managing tiered memory in modern systems, each with its own limitations. The first is in the OS kernel [29, 31, 37, 42, 46, 56, 59], and has been widely adopted in production for years. Most OS-based approaches rely on page faults [29, 32, 37, 58], access-bit sampling [33, 42, 48, 58], but both have fundamental drawbacks. Specifically, these approaches use access pattern detection based on OS pages or hardware information. However, as Python objects are typically smaller than a page, such a method leads to making decisions based on coarse-grained information. Moreover, these techniques normally do not balance well between tracing accuracy and overhead, sometimes even making the overhead unbounded [33, 57, 58]. Yet, they fail to provide easy-to-use software interface for users to adjust trade-offs. In addition, existing Python applications in the cloud typically have different priorities and Quality of Service (QoS) requirements, OS-based systems may give suboptimal performance [37] as they cannot be customized for different user-level programs.

The second type of tiered memory approach is tightly coupled with runtimes, which either define a new programming model offering self-defined APIs [16, 23, 49] in languages that support native execution (*e.g.*, C/C++), or leverage garbage collection in modern memory-managed languages to place program data [36, 54, 55, 60]. 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 (EMT) for memory-intensive Python programs. The key idea of EMT is that the *Reference Counting (refcount)* can be leveraged to infer Python object access patterns.

Our contributions are as follows:

- *Lightweight, sampling-based memory tracking*: Introducing a CPython-integrated approach that efficiently tracks object access patterns with controlled overhead (Sec. 5.1 & 5.2).
- *Inference of object hotness using refcount*: Accurately inferring memory access information by leveraging Python's built-in garbage collection metadata (Sec. 5.3).
- *Software-defined page table and adaptive migration*: Introducing an adaptive lazy demotion strategy and software-defined page table to enable seamless data movement between memory tiers (Sec. 6).

By integrating EMT into real-world memory-intensive Python workloads, we demonstrate improvements in memory efficiency and scalability. Using multiple real-world workloads in an emulated CXL-tiered environment, our evaluation shows that EMT outperforms existing solutions in most cases. Across 33 comparisons (11 workloads, each with three-tier configurations), EMT outperforms TPP[37] in 25 cases, AutoNUMA [3] in 30, and MEMTIS [33] in 15. In addition, EMT offers configurable trade-offs to balance performance, memory efficiency, and overhead, making it well-suited for cloud-based and HPC environments. Most importantly, EMT does not require any OS changes and can be easily integrated into cloud orchestration tools like Kubernetes [4]. We anonymously open-source EMT at https://anonymous.4open.science/r/cpython-20F7.

2 BACKGROUND

This section first discusses the CXL memory system, the protocol on which our platform relies (Sec. 2.1). Next, we provide background on CPython, the most widely used Python interpreter, and its garbage collector design, on which EMT is based on (Sec. 2.2).

2.1 CXL Memory System

CXL is an open standard interconnect designed to enhance the performance of data centers [14]. One of the key components of CXL is CXL.mem, which allows direct attachment of memory devices to the CPU. It uses PCIe's interface with custom link and transaction layers for low latency. The CXL.mem provides a unified memory space, enabling seamless memory expansion and reducing bottlenecks associated with traditional memory hierarchies, such as limited DRAM capacity, high latency in accessing far memory, and bandwidth constraints between memory tiers. By integrating CXL.mem, data centers can achieve higher memory capacity and more efficient resource utilization, thus addressing the growing demands of today's data-intensive applications. Since CXL latency is relatively high – almost double as compared to local DRAM – EMT leverages CXL.mem as a slow tier to offload cold data into CXL, monitors and detects when the data becomes hot, and moves the data to local DRAM to mitigate latency penalty.

2.2 CPython Runtime and Garbage Collector

CPython [27] is the standard implementation of Python Interpreter written in C and maintained by a large community. 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 environment.

In CPython, objects are created, managed, and destroyed through a well-defined structure called PyObject. 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. EMT's tracing insight comes from reference counting. Although refcounts do not directly indicate accesses to PyObjects, one can infer their real hotness by observing how refcounts are changed throughout the execution (Sec. 5.3).

In addition to refcount, CPython's GC periodically scans certain objects that reference each other but are no longer reachable from outside [44] (*i.e.*, cyclic-GC). It does this by maintaining three generational lists of container objects, such as lists, dictionaries, and classes. Objects that remain reachable are restored to their original state, while unreachable objects, identified as part of a cycle, are deallocated to free memory. EMT leverages CPython's existing cyclic-GC module to obtain all live PyObject references through recursive tracing (Sec. 5.1).

3 MOTIVATION

The design of EMT is based on the following motivations: limitation of adapting to QoS needs of workloads, and difficulty in balancing overhead and tracing accuracy in OS-based solutions. We present a summary of comparison with prior work in Table 1. In the remainder of the section, we investigate these factors in more detail.

3.1 Adapting to Workload-specific QoS Needs

OS-based memory tiering systems [29, 31, 37, 42, 46, 56, 59] manage system-wide memory but lack the flexibility to adapt to the specific QoS needs of individual applications. 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. Previous work like HeteroOS [28] and AutoNUMA [3] highlights the inefficiencies of OS-level policies that overlook application-specific QoS needs, leading to suboptimal memory placement.

AIFM [49] allows *application developers* to deploy custom logic on far memory and tune object structures to manage memory overhead. In contrast, EMT empowers *cloud providers* to enable or disable its tiering mechanism for Python code snippets through a simple API (listing 1), allowing users to adjust parameters to balance the trade-offs between tiering benefits and tracing overhead.

Insight 1. Rigid OS-based tiering solutions are not suitable for the dynamic needs of workloads. A better framework should offer opportunities to customize tiering configurations.

3.2 Balancing Tracing Accuracy and Overhead

State-of-the-art OS-based memory tiering solutions often struggle to balance accuracy and overhead in tracking memory access.

Table 1: Comparison of tiered memory systems focus on important features of some state-of-the-art tiering systems. Page fault-based tracking adds latency and lacks fine-grained (sub-page) accuracy, while access bit sampling struggles with scalability. System-wide deployments hinder workload-specific QoS customization. Prior runtime-level solutions focus on C++ or Java, and neglect Python. In contrast, EMT targets Python applications, supports sub-page (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 [37]	Page fault	Static access count	✗	✗	✓	✗	DRAM/CXL
AutoNUMA [3]	Page fault	Static access count	✗	✗	✓	✗	DRAM/DRAM
DAOS [42]	Access bit sampling	Static access count	✗	✗	✓	✗	DRAM/ZRAM
MEMTIS [33]	HW-based sampling	Memory access distribution	✓	✗	✓	✗	DRAM/NVM/CXL
AIFM [49]	Smart pointer interposition	CLOCK replacement	✓	✗	✗	✓	Local/far memory
TrackFM [51]	Smart pointer interposition	CLOCK replacement	✓	✓	✓	✗	Local/far memory
Write-Rationing [2]	GC interposition	GC-guided	✓	✗	✓	✗	DRAM/NVM
Panthera [53]	Static analysis + GC interposition	Static allocation + GC-guided	✓(but coarse-grained)	✓	✓	✗	DRAM/NVM
EMT (ours)	Refcount-based inference	Memory access distribution + eagerness-driven	✓	✗	✓	✓	DRAM/CXL

Accuracy. Techniques relying on page faults [3, 37, 58] and page table scanning [42, 48] face scalability challenges as memory footprint increases. For example, DAMON [42] attempts to reduce monitoring overhead as memory increases by grouping contiguous pages into a region. This approach assumes uniform access frequencies within a region, leading to inaccuracies. Experiments in MEMTIS [33] have shown that such coarse granularity can erroneously group pages with distinct access patterns, resulting in suboptimal data placement. On the other hand, methods based on Processor Event-Based Sampling (PEBS) [33, 46, 48, 58] scale with the number of memory accesses. It could miss accesses without triggering a lot of sampling overhead. EMT, on the other hand, can achieve high-resolution *object-level* access tracing.

Overhead. Scanning page table is expensive for high-accuracy tracing, as frequent and high-resolution sampling introduces significant CPU overhead. Meanwhile, PEBS incurs fundamental costs from PEBS assist, cache pollution, and interrupt handling [1]. Additionally, both page table scanning and PEBS lack any software interface to balance the trade-offs dynamically, which makes them less adaptable for diverse workloads in cloud and HPC environments. In contrast, although EMT cannot completely devoid overhead, it allows cloud providers to easily balance tracing accuracy and overhead with a *straightforward software interface*.

Insight 2. A good tiering system should achieve both fine-grained data access tracing and *controllable* overhead, in which current solutions fall short.

4 EMT DESIGN

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

Listing 1: EMT Python Interface

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

EMT's primary target is cloud providers who intend to provide cloud services backed by tiered memory. For example, on serverless platforms, providers can embed EMT within Python environments, enabling them to manage clients' code execution within a tiered memory system with tailored parameters, all while remaining transparent to end-users.

This simple API enables EMT for future QoS-aware orchestrator design. For example, Kubernetes Memory Manager [4] could prioritize Python workloads by adjusting tracing parameters: latency-sensitive jobs may delay tracing (via higher *mem_pressure*) to reduce overhead, while background tasks can be throttled (via higher *sample_interval*) until fast-tier resources free up. This dynamic, tunable tracing strategy optimizes resource utilization and enhances overall efficiency. However, such customization is not possible with OS-based tiering approaches [37].

4.2 Challenges

To achieve object hotness tracing for migration in tiered memory for Python applications, EMT 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 [22, 49]. However, this method does not apply to CPython virtual machine, which defines object structures in C[1]. The second one focuses on JVM-based languages (*e.g.*, Java, Scala) that

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

instruments read/write barriers [2, 54, 55]. 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, EMT leverages **reference counting** [10], 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 EMT. Instead, EMT 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 [44], EMT 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 [19], 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 [44], 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).

4.3 Pipeline

Figure 1 shows EMT's pipeline. Its principle is to periodically mark all live objects, observe refcount changes, and apply migration strategies accordingly. When enabled, EMT runs activates upon detecting local memory pressure. First, in the marking phase (Sec. 5.1), all live objects are marked, generating PyObject temperature metadata. Next, EMT 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, EMT 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). Built on a software-defined page table and guided by an adaptive lazy demotion policy (Sec. 6.2), the migration plane employs bucketing to classify hot and cold pages (Sec. 6.1). EMT keeps hotness statistics up to date through periodic cooling and adaptively minimizes tracing overhead by monitoring CPython cyclic-GC behavior during run time. Note that all EMT'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.

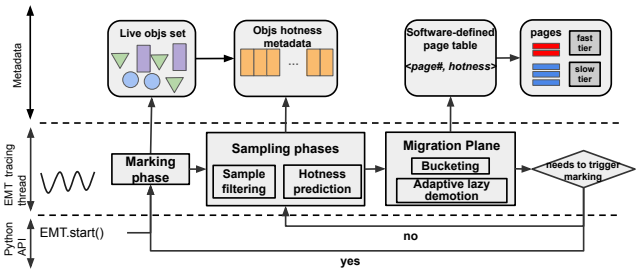

Figure 1: EMT Pipeline.

5.1 Marking Phase

To begin, EMT needs to know the scope of objects to sample. We noticed that CPython native cyclic-GC module maintains three generational lists to detect cyclic references [44]. 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 EMT solely samples on them.

To correctly mark all live PyObjects, EMT 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, EMT 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 (line 7, 17).

EMT'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, EMT must balance marking overhead with accuracy. Fortunately, this can be inferred by observing CPython's cyclic-GC behavior. Figure 2 shows the correlation between cyclic-GC counts (x-axis) and newly identified PyObjects (y-axis) in the same sampling period for several workloads. Few objects are not worth a marking (bottom left side) if EMT observes only a few cyclic-GC events occur.

To further mitigate overhead within one marking phase, EMT makes the following optimizations. First, it keeps track of the *length* and the *accumulated depth* of the current container PyObject being traversed. Once either of these two values meets the hard thresholds, the recursive function immediately returns. Figure 3 shows by confining the *length* and *depth*, one can dynamically control the trade-off between marking accuracy and overhead. Second, EMT maintains a `global_set` to record those already-traversed objects on the fly (lines 7, 18 in Algorithm 1), so that any PyObjects that are already in the set are not necessary to be recursively traced again. This is necessary since any PyObject can be referenced by one or multiple container PyObjects.

5.2 Sampling Phase

Having the scope of interested PyObjects during marking, EMT enters the sampling phase to sample their refcount changes. EMT leverages the system's endianness: it uses the unused 32 most significant bits (MSB) of the existing ob_refcnt field as a $growth$ counter,

Algorithm 1: EMT 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)
```

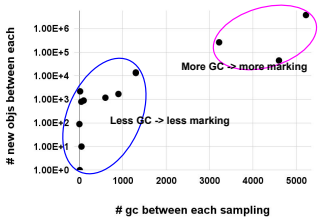

Figure 2: Marking phase frequency can be estimated by cyclic-GC frequency (x-axis) during run time.

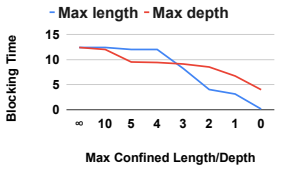

Figure 3: Application blocking time by confining different lengths and depths.

which **increases** regardless of whether the original refcount rises or falls.

During marking, EMT 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*. EMT 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 5x, even users do not intend to enable EMT during run time. This is expected to be higher than interposing JVM's read/write barriers, which directly indicates objects reads/writes because refcount in CPython occurs far more frequently than JVM barriers.

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

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 normally happens in modern workloads) takes around 1 second — hurting tracing recency. To mitigate this, we filter certain object samples under the following cases. First, EMT 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, EMT immediately marks those PyObjects as dropped to prevent further access.

5.3 Object Temperature Prediction

We conduct extensive *offline analysis* to correlate refcount changes with the real temperatures obtained from the OS, using DAMON [42]. Initial empirical analysis reveals a typical hotness pattern: *frequent but small refcount changes often correspond to higher temperatures.* DAMON uses access-bit sampling to determine page temperatures, as shown in Figure 4. But since it samples at the OS page level or page groups, its ground truth is inherently coarse. To infer object-level hotness from this data, we collect DAMON's ground truth using the highest sampling resolution and shortest interval, and run EMT with high sampling frequency on the same program. We then align EMT object addresses with the nearest page addresses in DAMON's output, given that a page temperature reflects all objects it contains.

For each object refcount change sample collected by EMT, we compute statistical features from refcount changes — such as median, standard deviation, number of non-zero changes, count of small changes, and the range. These features, along with their real hotness as labels, are sent to several prediction models, ranging from simple multiple linear regression (MLR) to more complex models like multilayer perceptron (MLP).

Table 2 summarizes the average performance of the models across various workloads using an 80/20 train-test split. Notably, all models performed similarly well in predicting PyObject temperatures based on refcount changes. For example, in the case of MLR, the selected features explained 70.9% of the temperature variance (R^2 = 0.709), and the predictions showed a strong positive relationship with actual temperatures (Pearson correlation = 0.842). Given the fact that more complex models, *e.g.*, Random Forest and

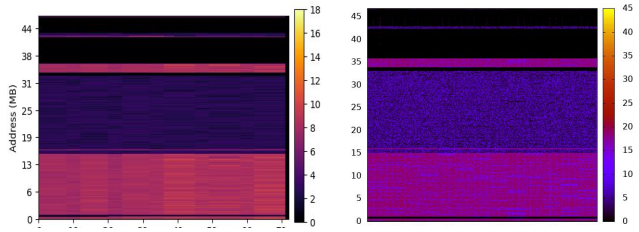

Figure 4: Heatmap generated from OS (right), and inferred from PyObject refcount changes (left).

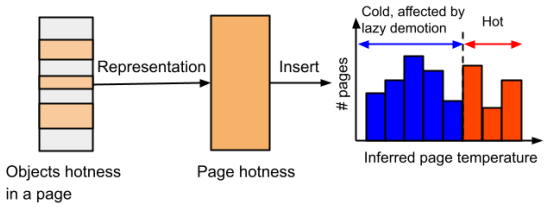

Figure 5: Software-defined page table contains represented hotness. We use bucketing to guide migration, while the hot/cold threshold is inferred by how eager hot pages are to be promoted.

MLP, provide marginal performance improvement but with higher inference overhead, we opted to integrate MLR into EMT.

Figure 4 illustrates the accuracy of the MLR model: the heatmap inferred from refcount changes (left) closely aligns with the one generated from OS page table entry scans (right) for the same Python matrix multiplication workload. Although absolute temperatures may differ due to normalization, the model can effectively distinguish hot and cold ones as long as their *relative* temperatures remain stable. This demonstrates that EMT can transparently capture object-level temperatures during run time without relying on OS-level information.

6 PAGE MIGRATION STRATEGY

After certain sampling phases, EMT triggers the migration plane. We choose to keep the migration unit at page level due to programming language constraints and object-level migration overhead, more details are discussed in Sec. 8. We use a software-defined page temperature table that maps PyObjects to their corresponding pages. Unlike the OS page table, our table is thinner, as it only tracks pages containing live PyObjects monitored by EMT.

Per page hotness representation. First, EMT aligns all sampled objects to their page boundaries and inserts the page addresses into the page table. During the insertion, EMT calculates each page's hotness using different representation models. We evaluate four models: Summed Hotness (SH), Median Hotness (MH), Interval Mode Hotness (IMH), and Averaged Hotness (AH) which compute a page's hotness using sum, median, interval mode, and average of all PyObject hotness values within that pages, respectively. For example, in Figure 5, if a page contains three PyObjects with hotness values of 3, 4, and 3, the SH of that page is 10. By default, EMT uses SH for simplicity and low overhead, but can be easily changed to other representations during build. More detailed analysis is provided in Sec. 7.3.

Next, EMT determines the threshold between hot and cold pages by applying bucketing and eagerness (Sec. 6.1). To mitigate unnecessary migration, EMT evaluates the eagerness of pages in the slow tier to be promoted (Sec. 6.2).

After that, EMT 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.

6.1 Page Bucketing and Threshold Determination

EMT 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 EMT, as explained below. With SH stored in the page table, the temperature of a page is affected by periodic cooling. In other words, if the SH of a page in the i^{th} interval is x, and the $(i-1)^{\text{th}}$ interval is $\text{SH}_{(i-1)}$, the SH in the i^{th} interval affected by cooling is as follows.

$$SH_i = k \times x + (1-k) \times SH_{i-1} \tag{1}$$

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

Using SH collected for page temperatures, EMT builds a histogram (Figure 5). In particular, the histogram consists of 10 buckets, and each bucket has a range of SH 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 SH range.

Determining hot/cold threshold for migration is challenging in EMT. 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 EMT — 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, EMT only tracks the virtual pages of the current Python program, so using the fast tier capacity to set this threshold does not make any 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 CPython 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. EMT 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 of

pages in the slow tier 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 (discussed in Sec. 6.2). $\text{Eager}_{\text{llc\_miss[i]}}$ 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 current LLC misses deviate from the expected range based on recent values. Similarly, $\text{Eager}_{\text{bw[i]}}$ is calculated by measuring memory bandwidth to the slow tier.

EMT uses this information to iterate through the histogram buckets, starting from index 0 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 $\text{Eager}_{\text{total[i-1]}}$, EMT decrements h, indicating that the current hot pages are less eager for promotion compared to the previous cycles.

6.2 Adaptive Lazy Demotion

Promoting hot pages to the fast tier makes sense when they are in the slow tier. However, immediate demotion of cold pages in the fast tier may be unnecessary, as they could become hot again soon. A simple solution is lazy demotion, where a page is demoted only if it is identified as cold in two consecutive samplings. This approach, used in TPP, helps reduce unnecessary migrations. Similarly, MEMTIS employs a 'warm bucket' where pages remain still temporarily. However, enabling lazy demotion regardlessly can have drawbacks, as it doesn't evaluate the trade-offs between migration overhead and benefits. In our experiment (see Figure 8), lazy demotion proves beneficial when the fast-tier RSS ratio is low, as it prevents unnecessary migrations and reduces contention for limited fast-tier resources. However, when the most amount of RSS is currently in the fast tier memory, disabling lazy demotion performs better. In this case, the benefit of migration outweighs the overhead.

Thus, to dynamically decide when to enable lazy demotion — particularly when the fast-tier allocation is uncertain (*e.g.*, 50%) — EMT reuses the *eagerness* statistics described earlier. If eagerness exceeds a predefined threshold T, lazy demotion is disabled; otherwise, it remains active. The threshold T is determined through a systematic, data-driven process using offline profiling across all workloads we evaluate in the next section. Execution times and eagerness vectors are collected during runs with and without lazy demotion, defining a range for eagerness values. We evaluate candidate thresholds in an increment of 0.1 * `eagerness_range`, overall 10 values. The final threshold is chosen as it minimizes the geometric mean of execution times, offering balanced performance improvement across all workloads.

Table 3: Workloads specification and their respective 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 insert random new items using foreign key.	3.6G
SQL2		8.8G
SQL3		3.7G

7 EVALUATION

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

- How does EMT perform with real-world memory intensive workloads compared to state-of-the-art tiering solutions in an emulated CXL environment (Sec. 7.2)?
- How do different hotness representations affect page migration (Sec. 7.3)?
- How effective is EMT's adaptive lazy demotion (Sec. 7.4)?
- What are the run-time and memory overhead for EMT (Sec. 7.5)?

7.1 Evaluation Methodology

Hardware setup. Our evaluation is conducted on a dual-socket machine, with one socket emulating CXL as the remote slow tier and the other serving as the local fast tier. The fast-tier socket features a 16-core Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz and varied 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 choose 11 representative memory intensive Python applications, including 3 SQL workloads using SQLAlchemy [11], and 8 graph computing workloads using the popular Python library NetworkX [24]. Table 3 shows detailed workloads description, including their memory consumption.

Comparison targets. We compare EMT against 3 systems: **TPP** [37], **AutoNUMA** [3], and **MEMTIS** [33]. We report the relative performance slowdown compared to the baseline in each workload runs entirely in the local fast tier without any tiering solution.

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 [43] 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. This simulates a real-world scenario where a server runs

multiple workloads, with our evaluation target (the workloads) and background noise (the memory hogger).

For EMT, 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, as recommended by its original paper. 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. To minimize variances, each workload in each configuration was run five times and we report the median value.

7.2 Performance Comparison

Figure 6 shows the average slowdown of all tiering solutions relative to running workloads entirely in the fast tier. Across 33 configurations (11 workloads × 3 memory ratios), EMT outperforms TPP in 25 cases and AutoNUMA in 30. While MEMTIS edges out EMT in 18 cases, EMT still achieves a lower geometric mean slowdown (1.18 vs. 1.20), highlighting its overall competitiveness.

Astar (Figure 6a) benefits from EMT 's MLR-based hotness inference via object refcounts, yielding 23.4%–25.5% lower slowdown than MEMTIS and up to 29.3% 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 are marginal, with slowdown only dropping from 1.55 to 1.45 as the fast-tier size increases from 25% to 75%. In contrast, AutoNUMA's slowdown decreases from 1.69 to 1.14. Comparatively, EMT maintains stable run-time variance across different fast-tier sizes while achieving the lowest slowdown.

In Bellman (Figure 6b), TPP has a relatively lower slowdown compared to Astar, as it better distinguishes hot/cold pages. AutoNUMA performs the worst due to minor page faults, with slowdowns ranging from 56.6% to 28.9% compared to the baseline. Similar patterns are observed in BFS_rand, KC, and SQL2. EMT still achieves the best result among all solutions.

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

SQL workloads (Figure 6i, 6j, 6k) populate relational databases with different table structures and insert random new items using foreign keys, but with different table scales and compute intensity. In this case, EMT behaves better than AutoNUMA by overall 4.7%. However, it is slightly slower than TPP by 5.1% and MEMTIS by 4.9% considering SQL2 and SQL3. Unlike in BFS_rand, this slowdown stems not from marking, but from EMT 's critical-path operations on the *growth* field, which are essential for refcount-based inference.

Finally, comparing TPP and AutoNUMA, TPP achieves an 11.2% geomean 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 fast tier. This minimizes the overhead of temperature detection. However, in workloads like Astar, Bidirectional, and SQL1 (Figures 6a, 6e, 6i), TPP's aggressive demotion leads to frequent migrations and higher overhead than AutoNUMA, even with lazy demotion enabled.

7.3 Page Temperature Representations Analysis

We implement and evaluate four object-based page temperature representations: Summed Hotness (**SH**), Median Hotness (**MH**), Interval Mode Hotness (**IMH**), and Averaged Hotness (**AH**). Directly comparing their effectiveness is difficult due to the coarse sampling of page hotness in DAMON, so we use the number of migrated pages — shown for Astar (irregular access pattern) in Figure 7 — as a proxy. Adaptive lazy demotion is disabled for clarity.

SH captures the **total activity** on a page, making it useful for estimating overall usage. Astar shows sporadic migrations triggered by concentrated bursts of object accesses. Over time, SH also shows a more gradual decline in migrations compared to other representations. We attribute this to two key factors. First, SH does not distinguish between a few very hot objects and many mildly warm ones, resulting in a relatively stable hotness distribution. Second, the same virtual address can represent different PyObjects at different times due to the managed runtime. Thus, it makes sense some objects accessed at different times are actually located in the same set of pages, reducing page migration. **MH** is **more robust to outliers** than SH, as it ignores extreme hotness values. However, it results in higher migration rates and greater fluctuation. When a few hot objects sit among mostly cold ones, the median remains low, causing unnecessary demotions. **IMH** shows the **most stable page migration** as it captures the most common object hotness level. However, it's sensitive to interval selection and may ignore important high-access cases. Lastly, **AH** shows the **most unstable migration pattern**, especially for irregular access workloads and highly skewed hotness distribution in each page. Given its volatility, AH is not recommended unless the access pattern is well-understood and predictable.

Note that except for **SH**, all other three representations require extra metadata for hotness calculation. For example, **MH** incurs an `O(log(N))` overhead per object hotness recording, where N is the average number of objects in each page. Performance of all four representations show marginal differences. This is because compared to page migration overhead, the stop-the-world in EMT's marking phase dominates the overall cost. However, this provides a takeaway for future research when trying to represent page temperature using objects within when there is no marking overhead like EMT.

7.4 Effectiveness of Adaptive Lazy Demotion

Figure 8 shows the normalized slowdown of five graph workloads across varying fast-tier memory allocation ratios (x-axis), where a y-axis value of 1 represents the best performance (least run time) for each workload, typically achieved when 75% of the RSS is in the fast tier. The workloads were run using EMT with three configurations: eager demotion, lazy demotion, and an adaptive mode that enables lazy demotion based on a profiled threshold T.

Eager vs. Lazy Demotion: When only a small portion of memory (e.g., 25%) is in the fast tier, lazy demotion improves performance by preventing unnecessary migrations of pages that may soon become

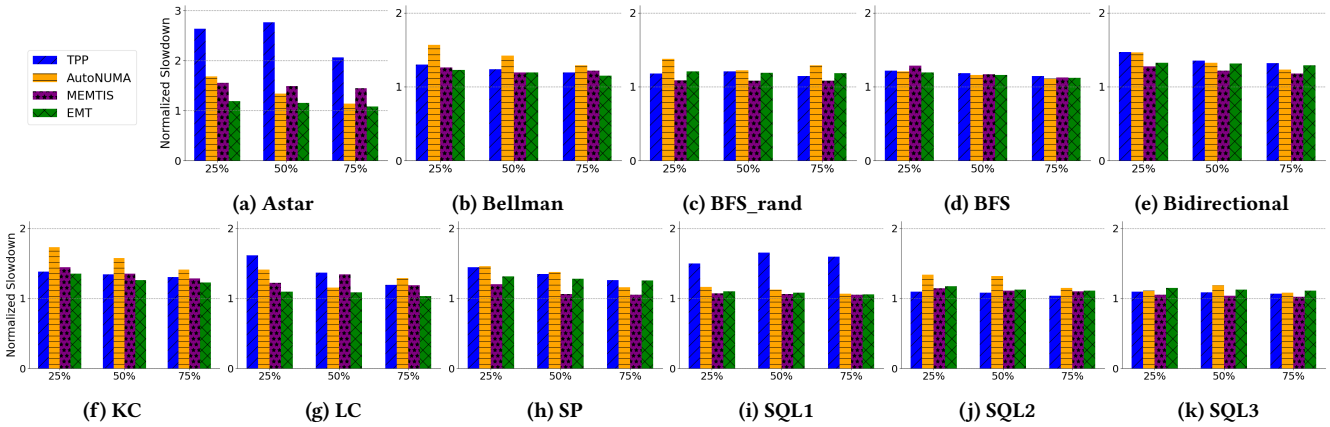

Figure 6: Workload slowdown ratios under different tiering configurations (with the fast tier set to 25%, 50%, and 75% of total RSS), compared to the baseline where each runs entirely in the local fast tier. Only MEMTIS is evaluated against the THP-enabled baseline. Across 33 comparisons, EMT outperforms TPP in 25 cases, AutoNUMA in 30, and MEMTIS in 15 cases.

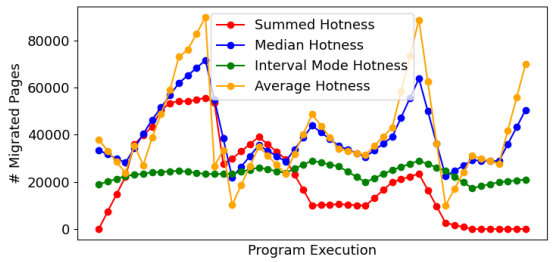

Figure 7: Number of migrated pages for different hotness representations in Astar.

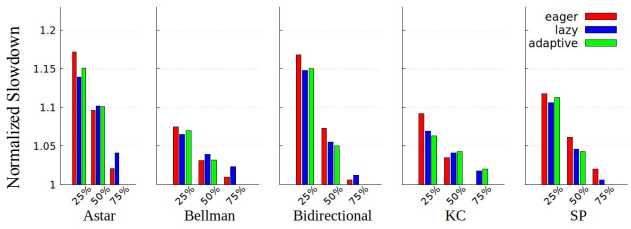

Figure 8: Effect of different demotion modes for graph workloads under varying fast-tier RSS ratios in EMT.

hot again. In such cases, eager demotion can cause inefficient "ping-pong" migrations, competing for fast-tier resources. Conversely, when most of the RSS is in the fast tier, eager demotion performs better (except for SP), as more frequent migrations face less competition and the benefits outweigh the overhead. For a 50% split across tiers, lazy demotion only outperforms eager demotion in Bidirectional and SP, indicating that more precise criteria for enabling lazy demotion are needed.

Adaptive Lazy Demotion: This mode balances performance by considering not just the RSS ratio but also factors like LLC miss ratio and slow-tier bandwidth during run time (Sec. 6.1). At 75% fast-tier allocation, adaptive mode performs best in 4 of 5 workloads (except KC). For 50%, it leads in 3 of 5 cases, and at 25%, it consistently performs between eager and lazy demotion. Overall, the adaptive mode is most effective at 75% fast-tier allocation, with slightly lower performance as the fast-tier size decreases (except for KC). This is likely due to our chosen T, which favors eager demotion more. Further online tuning of the threshold could improve performance.

7.5 EMT Overhead Analysis

Run-time overhead: EMT's overhead stems almost entirely from its marking phase, which accounts for over 99% of total overhead. Figure 9 shows the marking time as a percentage of the total workload run time across different fast-tier ratios, providing insight into EMT 's performance when excluding migration benefits.

Marking phases can contribute up to 8.3% (SP) of the total run time. This overhead is dynamically controlled by skipping unnecessary marking phases based on GC frequency and limiting the length and depth of each marking. Without these optimizations, the overhead would likely be higher. Most overhead occurs during the initial marking phase, which is essential for identifying PyObjects.

For all workloads, the ratio of marking time increases as fast-tier size grows. This is because the absolute marking time (numerator) remains stable while workload run time (denominator) decreases as more RSS is allocated to the fast tier, leading to larger division results. However, this ratio does not grow unbounded, as EMT only triggers marking when fast-tier capacity is insufficient.

Interestingly, fast-tier scarcity has a smaller impact on marking overhead than anticipated. For example, when fast-tier size increases from 25% to 75%, the overhead ratio increases by only 1% at most (in Astar). This suggests that EMT 's marking overhead is not significantly impacted by fast-tier pressure.

Memory overhead:

In Figure 9 (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

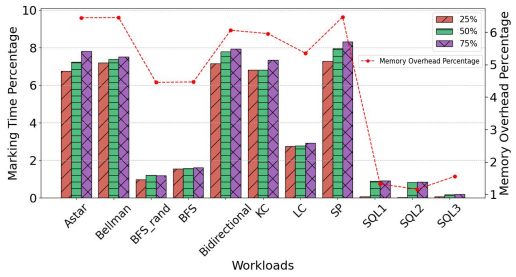

Figure 9: Marking time and memory overhead percentage for EMT. Marking time consists at most 8.3% of total running time, while memory overhead consists of at most 6.4% of total RSS.

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, EMT '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, more compact data structures could reclaim space once objects are repeatedly cold—an avenue for future work. Importantly, EMT embeds the *growth* field into unused bits of *ob_refcnt*, avoiding any memory overhead when tracing is disabled.

8 LIMITATION AND DISCUSSION

Inability to track native executions. Even though we put significant effort realizing EMT and minimizing overhead, EMT can only track PyObjects with refcount changes. Hence, it misses native objects from libraries like NumPy, SciKit-Learn, or TensorFlow [6, 18, 39, 52]. A potential solution is detecting native execution and delegating migration to OS-level methods.

Guidance on EMT vs. MEMTIS. While MEMTIS is more effective in some scenarios according to our experiments, it operates at a coarse-grained, at huge pages, and system-wide scope, limiting flexibility for workload-specific tuning. In contrast, EMT, as a user-level solution, provides fine-grained, per-application, per-code-segment control, and safe deployment without kernel changes.

We stress that EMT and MEMTIS (or other future high-performant OS solutions) are orthogonal to each other and can coexist. For example, both can be deployed in separate VMs on the same server: MEMTIS handles general tiering, while EMT is customized for VMs running Python applications with specific QoS Needs.

Why EMT uses page-level migration. We do not incorporate object-level migration for two principal reasons. First, existing object-level migrations (e.g., Write-rationing [2], Panthera [53], Semeru [54]) — originally devised for the JVM — are incompatible with CPython which does not move objects by itself, making it infeasible to bind GC-managed regions to DRAM/CXL and depend on automated migration. Adapting a CPython-specific object-level migration mechanism requires substantial changes to the existing GC and memory management schemes, making it impractical.

Second, even if this challenge is solved with engineering effort, forcibly enabling object-level migration will still cause substantial overhead. Preliminary work in SemSwap [13] reveals that consolidating hot objects into dense pages minimizes network traffic. However, it introduces considerable overhead in metadata management and run-time address translation. Furthermore, DiLOS [61] shows that page migration imposes comparatively 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 packed.

GC-agnostic design. EMT only uses CPython's GC for liveness marking, unlike Memliner [55] which modifies JVM GC internals. This decoupled design simplifies integration and avoids heavy CI/CD dependencies.

9 RELATED WORK

Given the exponentially growing memory needs, a number of works have explored tiered memory systems, which mainly fall into two categories, OS-based and software-based.

OS-based tiered memory falls into two categories. The first uses page table tracing [3, 17, 28, 29, 37, 42, 48], relying on page faults or access-bit sampling. It periodically checks and resets access bits, with overhead scaling to memory size due to page scanning. The second leverages hardware features like PEBS (Intel) and IBS (AMD) to capture exact memory addresses [9, 17, 40, 46, 48], avoiding page scans. However, high memory traffic increases sampling overhead [33].

Software-based tiered memory has been explored within application-level [26, 35, 47, 53] and library-level systems [16, 22, 40, 46, 49, 51] that offer fine-grained object tracking across caches, local tiers, and far memory. While MaPHeA [40] uses profile-guided heap allocation, its offline nature limits adaptability. AIFM [49] introduces remotable pointers but requires predefined APIs. Mira [22] and TrackFM [51] automate placement via compilers, yet need source access.

In managed runtimes, HCSGC [60] modifies ZGC for hot/cold segregation; Panthera [53] combines offline profiling with online GC for layout control. Semeru [54] and MemLiner [55] aim to reduce GC-induced far-memory overhead. Write-Rationing [2] prioritizes NVM endurance, while TeraHeap [30] reduces serialization overhead when offloading long-lived objects. Though some challenges overlap with CPython, EMT focuses on reducing performance loss when using CXL-based memory expansion.

10 CONCLUSION

In this work, we present EMT, a CPython runtime extension that enables transparent and efficient memory tiering for Python applications. EMT infers object hotness via refcount changes using a lightweight model, and manages migration through a software-defined page table with page bucketing and adaptive lazy demotion. Experiments show that EMT achieves competitive performance against state-of-the-art OS-based tiering in identifying and migrating hot data. Additionally, it allows cloud providers to tailor tiering strategies to meet diverse workload QoS requirements.

REFERENCES

[1] 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.

[2] 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.

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

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

[5] 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/

[6] 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.

[7] TIOBE Software B.V. 2023. TIOBE Index - TIOBE. https://www.tiobe.com/tiobe-index/. Accessed: November 21, 2023.

[8] Stephen Cass. 2023. The Top Programming Languages 2023 - IEEE Spectrum. https://spectrum.ieee.org/top-programming-languages-2024. Accessed: November 21, 2023.

[9] Jinyoung Choi, Sergey Blagodurov, and Hung-Wei Tseng. 2021. Dancing in the dark: Profiling for tiered memory. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 13–22.

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

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

[12] Maintainer Gabor Csardi. 2013. Package 'igraph'. *Last accessed* 3, 09 (2013), 2013.

[13] 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.

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

[15] Django. 2019. The Web framework for perfectionists with deadlines | Django. https://www.djangoproject.com/.

[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] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 727–741. https://doi.org/10.1145/3582016.3582031

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

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

[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] 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. https://doi.org/10.1145/3600006.3613157

[23] 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.

[24] 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).

[25] 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.

[26] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 875–890.

[27] Justapedia contributors. 2022. Python (programming language) — Justapedia, The Free Encyclopedia. https://justapedia.org/index.php?title=Python_(programming_language)&oldid=1119573205. [Online; accessed 19-October-2024].

[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. https://doi.org/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. https://doi.org/10.1145/3575693.3578835

[35] Zhe Li and Mingyu Wu. 2022. Transparent and lightweight object placement for managed workloads atop hybrid memories. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 72–80.

[36] 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.

[37] 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.

[38] A Lakshmi Muddana and Sandhya Vinayakam. 2024. SQLite3. In *Python for Data Science*. Springer, 201–216.

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

[40] Deok-Jae Oh, Yaebin Moon, Eojin Lee, Tae Jun Ham, Yongjun Park, Jae W Lee, and Jung Ho Ahn. 2021. MaPHeA: A lightweight memory hierarchy-aware profile-guided heap allocation framework. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 24–36.

[41] Pallets. 2010. Flask documentation. https://flask.palletsprojects.com/en/3.0.x/.

[42] 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.

[43] 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

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

[45] PyTorch. 2023. PyTorch. https://pytorch.org/.

[46] 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.

[47] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2021. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 598–611.

[48] 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.

[49] 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

[50] 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

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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. https://doi.org/10.1145/3503222.3507731

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.