Extended Abstract: Towards Efficient Python Interpreter for Tiered Memory Systems

Yuze Li
Virginia Tech

Shunyu Yao
Virginia Tech

Jaiaid Mobin
Rochester Institute of Technology

M. Mustafa Rafique
Rochester Institute of Technology

Dimitrios Nikolopoulos
Virginia Tech

Kirshanthan Sundararajah
Virginia Tech

Huaicheng Li
Virginia Tech

Ali R. Butt
Virginia Tech

Running data-intensive applications on modern tiered memory systems requires accurate and efficient data access tracing and migration to effectively use memory resources (e.g., TMO [14], AutoNUMA [2], AutoTiering [9], X-Mem [5], HeteroOS [8], HeMem [12], INFINISWAP [6], AIFM [13], Mira [7]). Prior solutions follow the principle of tracking data access frequencies (temperature) and automatically migrating them among tiered memory devices. They have taken either of the two approaches. The first type of solution transparently migrates memory pages (or hardware cache line for new protocols like CXL) between tiered memory devices [1, 2, 6, 8–10, 12, 14]. They require no code changes, but experience operation overhead and inaccuracies. Plus, they can only detect data temperatures at the page level or for groups of pages. Access information for data objects smaller than a page size cannot be used to determine page temperature, resulting in a coarser-grained temperature calculation. The second group defines new programming models to explicitly place/move fine-grained data objects. This type is tightly coupled with runtime, which either offers new programming models through self-defined APIs [5, 11, 13], or uses profiling and static analysis to explicitly guide data placement and migration [7]. However, they require either non-trivial application-programmer/library-writer efforts, or exhaustive profiling to reach the optimal placing strategy. Finally, none of the existing methods is readily portable to the popular language, Python, considering Python's top-ranking position in 2023 [3, 4].

We argue languages implemented as interpreters are ideal layers to track object temperatures, thus offering transparency to programmers and portability in the cloud. This WiP proposes Pypper, a module built in CPython to trace and migrate Python objects among tiered memory devices. We outline two major challenges in building Pypper.

Challenge 1: Tracking Python object temperatures. For languages such as C++, object temperatures can easily be obtained by overloading C++ smart pointers and the dereference operator (i.e., ->) and marking some bits as hotness indicators [7, 11, 13]. However, such a strategy cannot be applied to CPython, whose runtime is based on C. Our major insight is, reference counting, a popular strategy for garbage collection, is not completely unrelated to how objects are accessed. Based on that, Pypper infers object accesses through the changing pattern of object reference counts (*i.e.,* active trace).

Challenge 2: Tracing overhead caused by Global Interpreter Lock (GIL). While Pypper is obtaining live object references (*i.e.,* live trace), it needs to acquire the GIL to block the application in the CPython critical path. The blocking time scales with the number of objects that are traced. We observed the live set of objects is not likely to change until a cyclic-GC happens. Therefore, Pypper performs a live trace only when the cyclic-GC is triggered. It further extracts information from metadata maintained by CPython's cyclic-GC module to selectively limit the scale of objects to be traced. All in all, Pypper comprises a tiered frequency for live trace, active trace, and migration, respectively, with each module decoupled and metadata synchronized.

Pypper targets Python applications and promptly traces and migrates PyObjects accordingly. Pypper's tracing design is orthogonal to any other existing memory technologies – it sheds light on tracing object access temperature within languages implemented as interpreters using reference count changes. Plus, Pypper offers trade-offs between tracing accuracy and performance.

For native execution that happens outside CPython runtime, the reference count-based approach fails to capture access information. Thus, we plan to build an inference model by collecting high-level system information and trace allocation.

Pypper instruments the CPython 3.12 release version to obtain live PyObjects reference count changes and infer their hotness. Pypper requires *no* Python program changes, only except for APIs calling to enable and disable Pypper module during runtime to efficiently run clients' Python applications in tiered memory systems. Only the code snippets encapsulated by API calls are affected. Pypper runtime design is independent of hardware/OS, providing a portable solution that can be readily deployed in today's cloud.

References

[1] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[2] Andrea Arcangeli. *AutoNUMA AutoNUMA Red Hat, Inc*. 2012.

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

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

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

[6] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[7] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692–708, New York, NY, USA, 2023. Association for Computing Machinery.

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

[9] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for {Multi-Tiered} memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 715–728, 2021.

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

[11] Yifan Qiao, Zhenyuan Ruan, Haoran Ma, Adam Belay, Miryung Kim, and Harry Xu. Harvesting idle memory for application-managed soft state with midas. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, April 2024.

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

[13] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, November 2020.

[14] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. Tmo: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 609–621, 2022.

Towards Efficient Python Interpreter for Tiered Memory Systems

Yuze Li[1] Shunyu Yao [1] Jaiaid Mobin [2] M. Mustafa Rafique [2] Dimitrios Nikolopoulos [1] Kirshanthan Sundararajah [1] Huaicheng Li [1] Ali R. Butt [1]

[1]Virginia Tech [2]Rochester Institute of Technology

Tiered Memory Systems

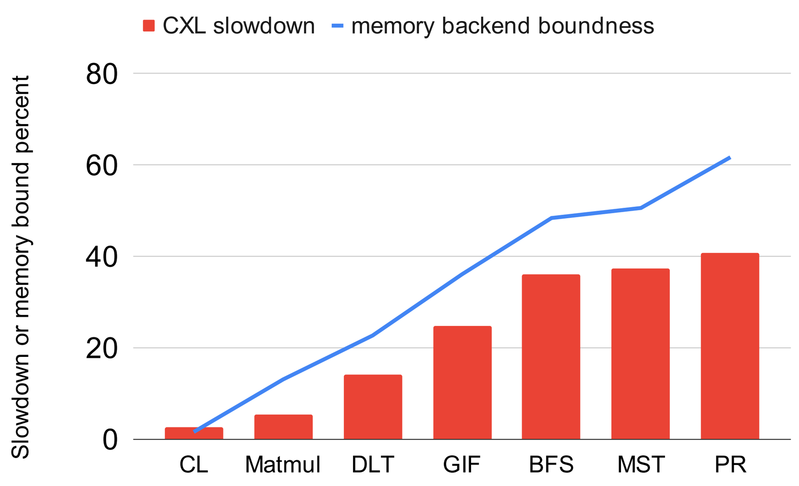

Figure 1. Slowdown percent of different workloads in CXL.

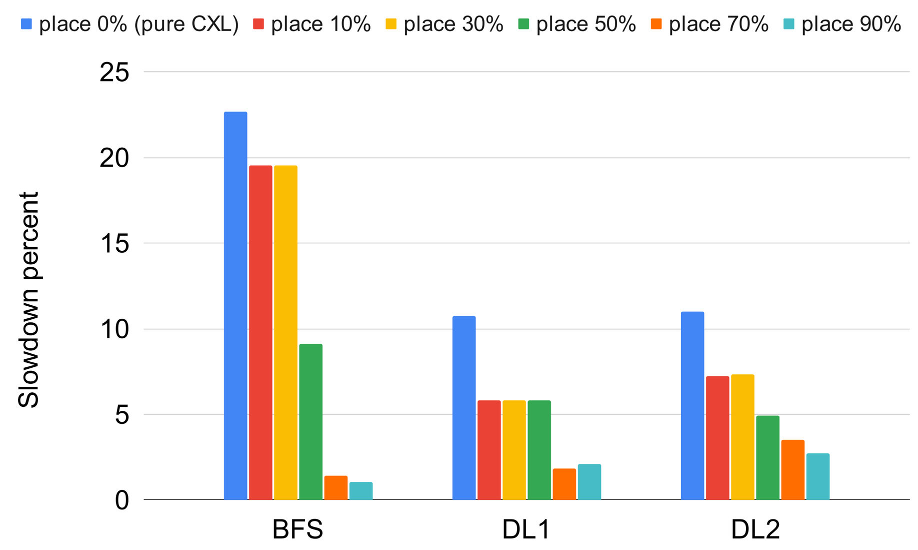

Figure 2. Workloads slowdown by static placing different percent of hottest memory pages to DRAM, the rest to CXL.

The emergence of low-latency non-DDR technologies offers cheaper $/GB memory cost. Running modern data-intensive applications in tiered memory systems experiences different percentage of slowdown. The principle is to track data access frequencies and automatically migrates them among tiered memory resources. Thus, a good solution must be:

- **Accuracy**: Be precise about the memory boundaries to be hot or cold.
- **Low overhead**: Solutions should not interfere with applications that much.
- **Portability**: Can readily be deployed to today's cloud.
- **Transparency**: No need for program re-writing, static analysis.

Problem of Existing Solutions

OS level: Page table entry checking, hardware event sampling, LRU, AutoNUMA, etc.

- **Coarse-grained observation point**: Sub-page information, and application semantics cannot be extracted.
- **Unbalanced accuracy and overhead**: By increasing the accuracy, overhead will increase

Runtime level: Defines new programming models through APIs, source code static analysis, and profiling.

- **lack of transparency**: Involves non-trivial programmer efforts, or exhaustive profiling.

None of the existing methods can be directly ported to the popular language, Python, considering Python's top-ranking position in 2023.

Challenges of Tracking Python Object Temperatures

Challenge 1: Method of Tracing

- Unlike C++, CPython does not offer smart pointer and operator overloading.
- Unlike JVM-based runtime, CPython does not have read-write barriers to instrument.

Challenge 2: Tracing Overhead

- CPython only maintains the references of **container** PyObjects, obtaining **all** PyObjects references requires the **GIL held** (application paused).

Challenge 3: Handling Native Calls

- CPython does not capture runtime semantics in native executions (C/C++).

Major Insights

Insight 1: **Reference counting** can be a potential indicator to infer PyObjects accesses (challenge 1).

Insight 2: The set of live PyObjects is **not likely to change** until a cyclic-GC is triggered; selectively tracing based on object semantics (challenge 2).

Pypper Overview

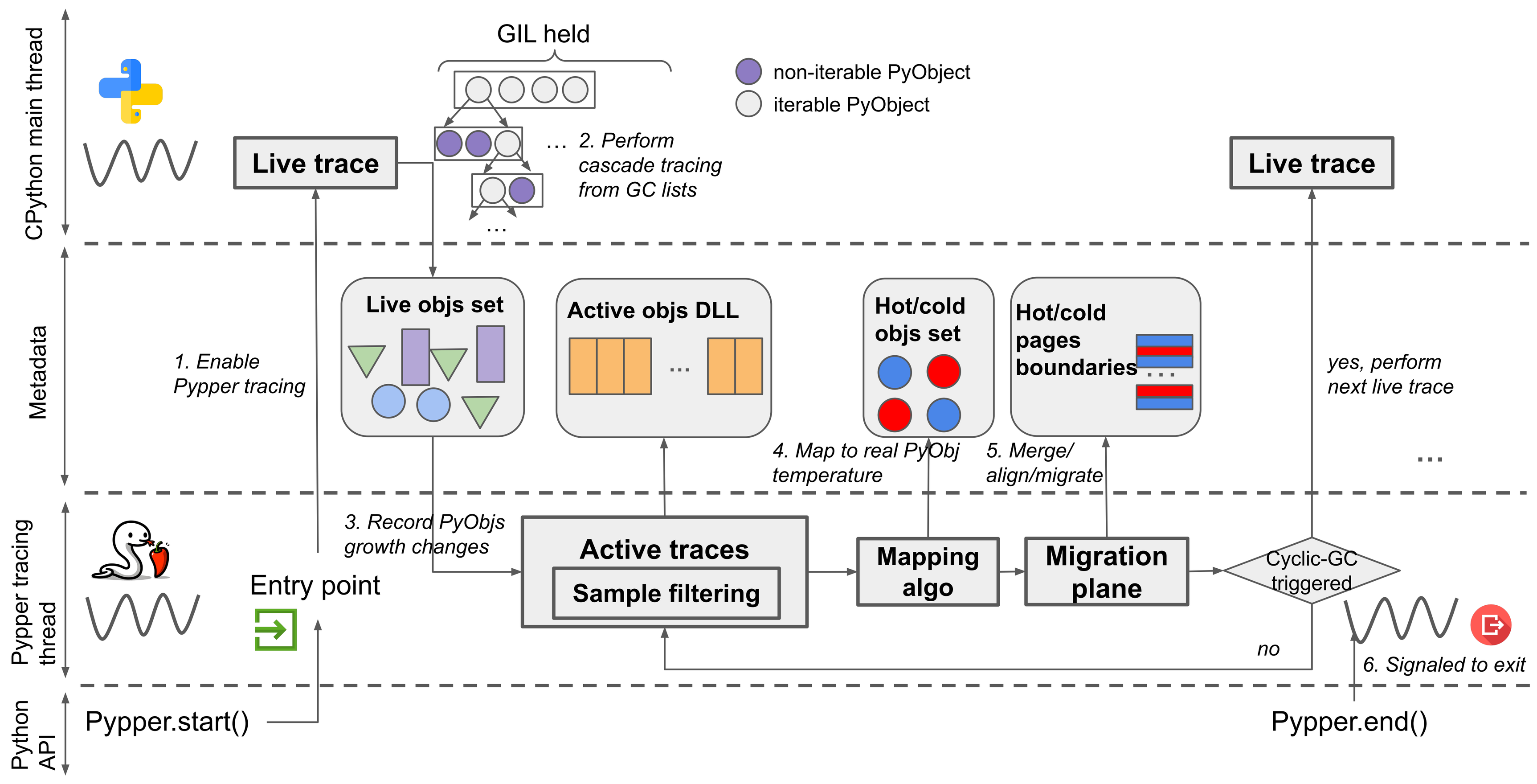

Figure 3. Pypper's workflow.

Pypper comprises a control layer and a metadata layer. The control layer populates and analyzes the metadata.

1. Invoked from Python API (`Pypper.start()`), tracing enabled within CPython main thread.
2. **Live trace** cascade traverses the cyclic-GC list to get all PyObjects references.
3. Pypper triggers a separate CPython thread for **consecutive active traces**, and records refcnt changes for each observed PyObject.
4. **Mapping algorithm** inspects the captured refcnt changes to infer the real PyObject temperatures.
5. **Migration plane** merges hot/cold objects into compact segregated memory ranges, aligns them to page boundaries, before migrating to designated areas.
6. Upon receiving stop signal (`Pypper.end()`), Pypper frees metadata, resets states, stops tracing.

Preliminary Results

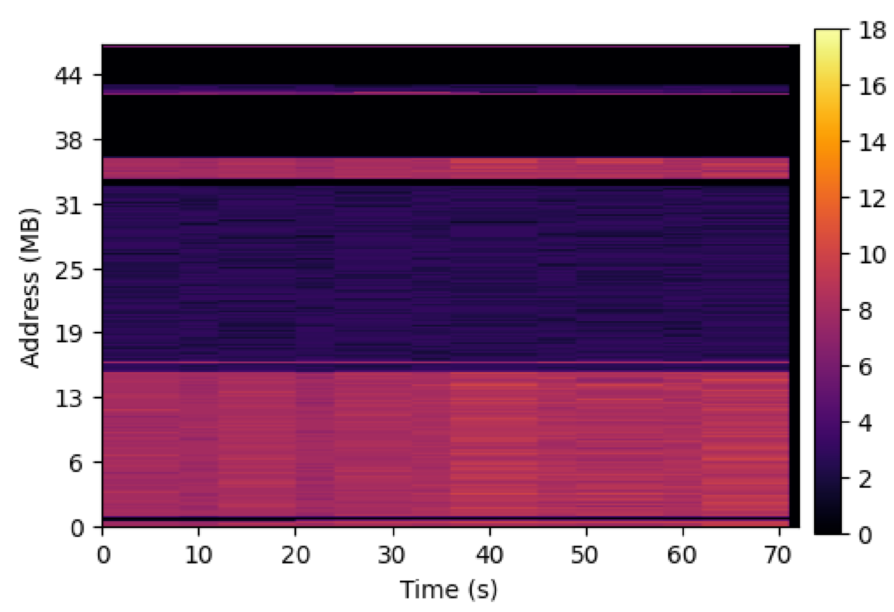

Figure 4. Inferred PyObj temperatures based on refcnt changes.

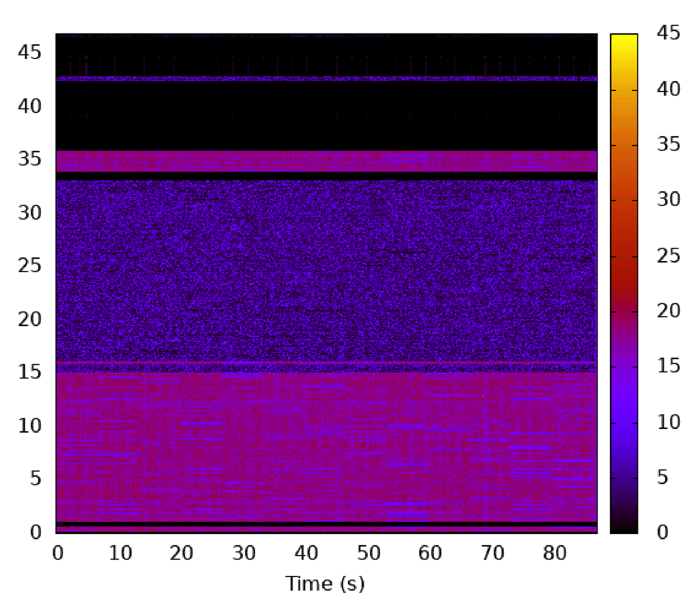

Figure 5. Real heatmap from OS-based profiling.

Takeaway: The reference counting in the GC scheme can also be used to infer object temperatures by defining a mapping model.

WiP and Future Work

Live Trace Overhead Mitigation (WiP)

- Make the best use of CPython's cyclic-GC module by only traversing **newly survived** container PyObjs.
- Filter live PyObjs by observing their semantics, e.g., length, depths.

Mapping Algorithm (WiP)

- A fine-grained mapping module from refcnt-changing to real object temperatures is yet to be defined.

Handling Native Executions (future work)

- Pypper should distinguish and handle native execution that is not based on refcnt changes.