Dancing in the Dark: Profiling for Tiered Memory

Jinyoung Choi
University of California, Riverside

Sergey Blagodurov
Advanced Micro Devices, Inc.

Hung-Wei Tseng
University of California, Riverside

***Abstract*—With the DDR standard facing density challenges and the emergence of the non-volatile memory technologies such as Cross-Point, phase change, and fast FLASH media, compute and memory vendors are contending with a paradigm shift in the datacenter space. The decades-long status quo of designing servers with DRAM technology as an exclusive memory solution is likely coming to an end. Future systems will increasingly employ tiered memory architectures (TMAs) in which multiple memory technologies work together to satisfy applications' ever-growing demands for more memory, less latency, and greater bandwidth. Exactly how to expose each memory type to software is an open question. Recent systems have focused on hardware caching to leverage faster DRAM memory while exposing slower non-volatile memory to OS-addressable space. The hardware approach that deals with the non-uniformity of TMA, however, requires complex changes to the processor and cannot use fast memory to increase the system's overall memory capacity. Mapping an entire TMA as OS-visible memory alleviates the challenges of the hardware approach but pushes the burden of managing data placement in the TMA to the software layers. The software, however, does not see the memory accesses by default; in order to make informed memory-scheduling decisions, software must rely on hardware methods to gain visibility into the load/store address stream. The OS then uses this information to place data in the most suitable memory location. In the original paper [1], we evaluate different methods of memory-access collection and propose a hybrid tiered-memory approach that offers comprehensive visibility into TMA.**

I. Introduction

Traditional DRAM memory cells are anticipated to become too small (somewhere below 10 nm) to reliably hold a detectable charge in the coming years. DRAM cells will thus reach their scaling limit—the point at which their cost can no longer be reduced through process shrinks. Cost reductions in DRAM-cell production will therefore stall and open the door for emerging, mostly non-volatile, memory technologies, including phase-change memory (PCM), 3D XPoint, memristor RAM (MRAM), Spin-Transfer Torque RAM (STTRAM), and fast flash memory (e.g., Z-NAND) [2]. The future landscape of main-memory architecture for data-center servers will inevitably require a *hybrid* approach as emerging non-volatile memory (NVM) technologies and DRAM host the growing working sets for a wide range of data-intensive server applications (e.g., in-memory key-value stores, databases, VMs consolidated on individual cloud servers, and high-performance computing [HPC] applications with large memory needs).

This paper adopts a tiered-memory architecture that maps all main-memory technologies into a single address space, and the system software (hypervisor, runtime middleware, or the OS) manages the mapping of underlying memory components. Compared to alternatives that treat DRAM as a cache for NVM [3] or treat NVM as a swap space for DRAM [4], [5], this architecture allows in-place updates to all memory pages, avoids data consistency issues, uses memory cells more efficiently, and leverages existing software support for non-uniform memory access (NUMA) architectures [6].

Due to the diverse latency, bandwidth, power, persistence, and cost/GB characteristics of memory technologies and their associated byte-addressable interfaces (e.g., NVDIMM-P, CXL, 3D XPoint DIMM, CCIX, and Gen-Z), successful tiered-memory architectures must rely on the system to minimize the latency associated with memory-content accesses and address translation for each memory request. Optimizing memory-access latency depends on the system's ability to capture frequently used, or *hot*, data and dynamically allocate that data to the fastest memory technologies available.

Unfortunately, optimizing system software's use of memory pages and TLB entries faces significant challenges:
(1) *Poor visibility.* Because hardware serves memory accesses, software is left in the dark about *how* memory is accessed.[1] The kernel only gains some visibility when a fault handler is invoked due to a missing memory page or a protection violation [7]. But faults rarely happen on a tiered-memory system (because no memory tier is exposed as a swap), and when they do happen, they incur significant overhead [8]. Further, identifying application-software code segments that most often access memory is complicated by multi-level instruction/data/TLB caches that exist between application threads and the actual memory accessed via cache/TLB misses.
(2) *Diversity of hardware monitors.* Software can gain some visibility into memory accesses through certain backdoor monitoring features exposed by hardware vendors. Moreover, each feature offers unique trade-offs with respect to verbosity, input sensitivity, and monitoring overhead. Studies are needed to evaluate these trade-offs for the monitoring methods currently available.
(3) *Creation of a single profiler metric.* Even if software can enable a given set of hardware monitors, there is still the question of how to process the information the software receives. A profiler should combine information from many monitoring sources to accurately and reliably identify hot/cold pages in the tiered (hybrid) memories (*cold* pages being infrequently

[1] After a miss in a hardware cache, the hardware data fabric delivers loads and stores to the hardware memory controller. Address-translation misses from hardware TLBs are also served by the hardware page-table walker (PTW) on x86 and ARM architectures. SPARC used to be an exception, with a software PTW and a hardware PTW cache.

accessed pages). Ideally, the profiler abstracts the many low-level monitoring details and presents the policy engine with a simple list of pages ranked by hotness. This ranking approach ensures a stable, vendor-agnostic profiler-policy interface so system software developers are free to handcraft their own hybrid memory-architecture policies independent of system-specific hardware configurations.

In the original paper [1], we evaluate multiple memory monitors and propose a unified approach that offers verbose, low-overhead visibility into tiered memory. We propose a *tiered-memory profiler* (TMP) as a solution—a profiler that leverages existing microprocessor support to lower software overhead and deliver more accurate results relative to existing memory profilers. TMP periodically retrieves raw memory-access-related data from underlying processors, aggregates the collected data, and produces meaningful statistics for memory-management policies. TMP is completely transparent and requires no modifications to applications; TMP simply uses analytics to help system software allocate memory.

The TMP-based tiered memory policies improve performance by up to 70% due to comprehensive profiling support.

The original paper [1] makes four key contributions:
(1) The paper presents a low-overhead, high-accuracy profiling mechanism that mitigates performance issues in TMAs.
(2) The paper generates insights that can guide efficient memory-management policies for TMAs.
(3) The paper implements and evaluates memory-management policies using the proposed profiling mechanisms to achieve speedups without any hardware modifications.
(4) The paper introduces an open-source tool as an upgradable solution to address performance in tiered memory systems.

II. TMP Design

Unlike prior profiler designs that relied on high-overhead software-based mechanisms or employ piecemeal monitoring hardware support, TMP leverages multiple key hardware features available across modern processors and achieves low profiling overhead. The TMP interface abstracts most low-level hardware details yet reveals verbose profiling statistics for optimizing memory-management policy decisions.

Figure 1 shows the architecture of TMP and TMP's interaction with other system components. We implemented TMP as a Linux prototype so we could easily adjust the parameters to optimize TMP's design. TMP includes a kernel-space module, a user-space daemon, and additional modules for A(Accessed)-bit, IBS (Instruction Based Sampling)/PEBS (Processor Event Based Sampling), and performance counter profiling mechanisms in Linux. The TMP driver manages and collects data from different software and hardware profiling mechanisms. It stores the data of a page by extending its page descriptor (PD) structure. The user-space TMP daemon runs concurrently with target applications to collect their process and thread IDs and receive configuration parameters, while, in return, producing meaningful statistics.

TMP's use of multiple, complementary monitoring methods maximizes informativeness and minimizes overhead. TMP takes advantage of trace-based profiling methods (IBS/PEBS) to inspect memory accessed from regular last-level caches (i.e., if the data source is out of local, combined level 3 LLCs). TMP supplements this information with the PTE's A-bit profiling to gain visibility into memory accesses from the TLB caches (a.k.a. cache misses of the address translation path). Additionally, TMP minimizes profiling overhead by enabling trace-based and PTE bit collection only when TMP sees increased accesses to the actual memory (not cache hits); to do this, TMP continuously monitors LLC and TLB miss rates via the HW counters, which incur minimal collection overhead when identifying periods of inactivity.

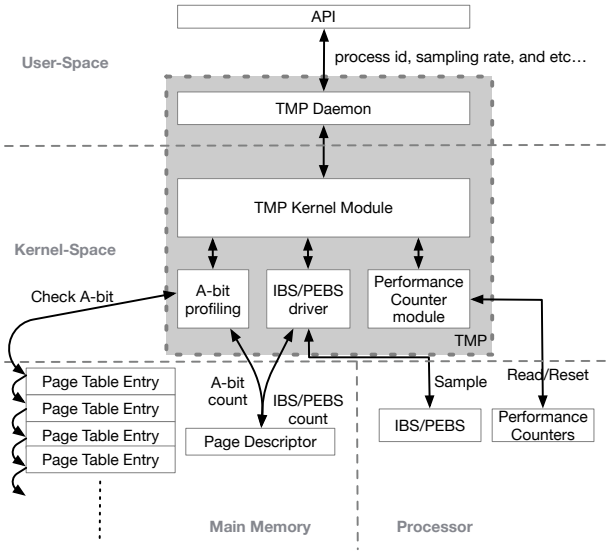

Fig. 1: The architecture of TMP

References

[1] J. Choi, S. Blagodurov, and H. Tseng, "Dancing in the dark: Profiling for tiered memory," in *IPDPS 2021 (accepted, not yet published)*.
[2] J. Handy and T. L. Group, "Emerging memories: Why now? opportunity opens for 3D XPoint, MRAM, and similar designs," 2019.
[3] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories," in *HPCA 2015*.
[4] D. Magenheimer, C. Mason, D. McCracken, K. Hackel, and O. Corp, "Transcendent Memory and Linux," 2006.
[5] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan, "Software-defined far memory in warehouse-scale computers," ASPLOS 2019, pp. 317–330.
[6] https://www.kernel.org/doc/html/latest/vm/hmm.html.
[7] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "BadgerTrap: A tool to instrument x86-64 TLB misses," *SIGARCH CompArch News*, vol. 42.
[8] M. Oskin and G. H. Loh, "A software-managed approach to die-stacked DRAM," PACT 2015, pp. 188–200.