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 this paper, 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 [1]. 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). 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 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 [2] or treat NVM as a swap space for DRAM [3], [4], 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 [5].

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 most performant 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 gains some visibility when a fault handler is invoked due to a missing memory page or a protection violation [6]. 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 [7]. 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. We describe the known memory-monitoring features in section II-B, but the features are often vendor-specific, with no standardization available. 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 accessed pages). Ideally, the profiler abstracts the many low-

[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.

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 this paper, 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.

This paper implements TMP in Linux, but the system designer can implement the same concept to collect data and generate hardware-based statistics to further reduce overhead. By evaluating data-intensive HPC and cloud applications on different processor architectures, we confirm that TMP reveals more information than existing memory-profiling mechanisms.

We demonstrate the effectiveness of TMP with page-placement policies derived from prior work. The TMP-based tiered memory policies improve performance by up to 70% due to comprehensive profiling support.

In summary, this paper 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 a profiling tool as an upgradable solution to improve performance in tiered memory systems.

II. Background

This section describes TMAs and justifies their adoption over alternatives; because TMAs rely more heavily on the system to judiciously use available memory technologies and underlying hardware support, TMAs require more information from a module/device like TMP. We also provide an overview of the profiling mechanisms that make TMP possible.

A. Tiered Memory and TMA

TMA maps all available physical memory locations into a single large address space. Theoretically, applications and system software can treat TMA as conventional DRAM without any modifications, but because hardware characteristics and performance differ significantly, a system using TMA typically categorizes its underlying memory technologies into different *tiers*, where upper tiers offer lower access latencies and higher bandwidth. In this paper, we refer to DRAM as *tier 1* memory technology and other NVM technologies as *tier 2* technologies. The tiered-memory system dynamically remaps and migrates memory from tier to tier in order to increase the fraction of memory accesses served from the faster memory tier.

Tiered memory is similar to NUMA in that all byte-addressable memory modules present on the system (irrespective of their media technology or topology) are exposed to the OS in the same physical address space. Memory in NUMA is closely aligned with multi-CPU placement and is therefore physically distributed across the server [8]. The NUMA API is meant to reduce the latency of memory access by pulling pages close to the CPU where the associated process is running [9]. Linux and Windows can enable *AutoNUMA* balancing for page portions (e.g., 256MB portions) by periodically changing page-table entry (PTE) permissions to *no access*. If an attempt is made to access an unmapped page, a page fault is generated and the kernel takes over and identifies the task that accessed the page. AutoNUMA then determines whether to move the page closer to the calling task or move the task closer to the page. With tiered memory, by contrast, all CPU cores are close to the fast DRAM rather than the NVM. Unlike with NUMA, the problem with tiered memory is not about localizing data near its compute; rather, the problem is that there may be more data than can fit in the local DRAM memory.

There are four advantages to using software-controlled tiered memory over using a *hardware cache*, which hides tier 1 memory from software via a giant last-level cache (LLC) [2].

First, tiered memory allows *in-place* memory access directly from the tier on which the data reside. With a hardware cache, the requested memory block is brought into the tier 1 cache from the tier 2 addressable memory, resulting in increased traffic. The in-place memory feature also differentiates the tiered-memory approach from the page-cache approach. With the page-cache approach (exposes tier 2 memory as a swap device [3], [4]), accessing a single cache line via tier 2 swap produces a costly page fault and is followed by the movement of an entire data block into addressable tier 1 memory [7].

Second, tiered memory avoids the significant hardware costs of large bandwidth-inefficient tags-in-DRAM (a.k.a. *cache tags*) [2], and contention issues from the directly mapped DRAM caches (and the related difficulty of using randomized free-page lists as a remedy) [10]. In TMA, a memory page can be found in either one of the tiers; caching would create duplicated, potentially inconsistent copies of pages in memory and would require mechanisms to keep data consistent.

Third, the tiered-memory solution allows for cache policy *fine-tuning* (via workload mixes, service-level agreements, etc.) to accommodate high-level policy decisions and eliminate excessive migration.

Fourth, with the similarity of entrenched NUMA architecture, tiered memory can leverage NUMA and Heterogeneous Memory Management (HMM) [5] system infrastructures. There are ongoing debates in the Linux community about how to expose NVM to the OS. Current proposals revolve around configuring NVM into CPU-less NUMA nodes and managing TMA balancing with AutoNUMA or other existing

NUMA methods [11], [12]. Here again the focus is mostly on how memory is allocated and moved, not on how hotness is profiled (e.g., how the periodic unmapping and page-fault handling in AutoNUMA incurs overhead [13]). Our work, which focuses on comparing various monitoring methods to gain maximum hotness visibility with minimum overhead, benefits both NUMA and tiered memory.

B. Memory Profiling Methods

Unlike hardware memory-scheduling mechanisms that can gain visibility into memory accesses by tracking individual cache lines [14] [15], system software (including the OS) is unable to see regular, unfaulted memory accesses by default. In order to determine which pages should be placed in fast versus slow memory across the tiered-memory hierarchy, the software needs help from the hardware.[2] Below we list available methods for the OS to gain such visibility without the need for profiled workloads to be modified or recompiled.

Page Table Entry (PTE) bit tracking. In most architectures, a PTE includes an accessed (A) bit and a dirty (D) bit. The OS can clear these PTE bits, and the hardware page-table walker (PTW) will set them. By observing the A-bit after each reset, the OS can determine whether the page was used at least once in the time interval since the last reset. The A-bit does not differentiate between a page that was accessed a single time and a page that was accessed multiple times over the course of the profiling interval. More frequent A-bit checks improve the informativeness of profiling but also increase overhead.

A-bits are used for performance tuning, and the PTW sets the A-bit on a TLB miss. On the other hand, D-bits are used for correctness (to evict to the backing store), and so they are part of a TLB entry; if the D-bit on a store is 0, the PTW sets the D-bit in the PTE regardless of TLB hit status [16].

Intel's Page-Modification Logging (PML) hardware feature automates D-bit collection. When PML is active, each write that sets a D-bit also generates an entry in an in-memory log with the physical address of the write (aligned to 4 KB). When the log is full, a notification to the system software is generated, and a hypervisor can specify an available set of log entries to monitor the number of pages modified by each thread [17]. In this paper, we focus on performance-oriented optimizations reflected in the A-bit.

Trace-based profiling (TBP). TBP enables the collection of address traces from load or store instructions. On AMD systems, Instruction Based Sampling (IBS) [18], [19] enables CPU instructions to be tagged as they traverse through the pipeline, allows data to be collected as the instruction executes, and raises an interrupt when the instruction retires. IBS op (execution) sampling uses every nth micro-operation (n configurable) and collects the following: virtual and physical data addresses for loads/stores, hit/miss latencies for the data-cache access status, and the TLB hit/miss/page size. IBS also generates a northbridge status for the load/store; that is, IBS (a) reveals whether it has been serviced by the northbridge in the same/remote memory node and (b) provides its data source (memory, local L3 cache, etc.). Software can adjust the sampling rate based on the observed IBS overhead.

Lightweight Profiling (LWP) [20] is an AMD64 hardware extension for Family 15h AMD processors that differs from IBS in that LWP collects large amounts of data before generating an interrupt. When enabled, LWP hardware monitors one or more events during the execution of user-mode code and periodically inserts event records into a ring buffer in the address space of the running process. When the ring buffer is filled beyond a user-specified threshold, LWP can produce an interrupt. The interrupt, in turn, causes the OS to signal a process to empty the ring buffer.

Intel's Processor (or Precise) Event Based Sampling (PEBS) is a trace-based feature similar to IBS/LWP in which the processor records tagged samples in a designated memory region. PEBS samples can be selected based on many events (e.g., cache misses), and each PEBS record contains the timestamp, the linear address, and the physical address, etc. [21].

Hardware performance counters (HWPCs) are special hardware registers available on most modern CPUs and GPUs as part of the Performance Monitoring Unit (PMU).[3] These registers obtain information about certain types of hardware events, such as retired instructions, cache misses, and bus transactions. PMU models from Intel and AMD can monitor hundreds of possible events covering many aspects of a given microarchitecture's behavior [23]. HWPCs can track these events without slowing down applications or the kernel [24]. The number of events that can be tracked in parallel depends on the availability of counter registers inside the PMU. Software tools like `perf` and `pfmon` can monitor more events than there are physical registers via event multiplexing.

Unlike other monitoring systems, HWPCs are coarse-grained—one metric for all process pages—and cannot be used to obtain a memory access trace [25]. Nonetheless, HWPCs can effectively detect elevated memory-usage phases when other profiling methods should be enabled. HWPCs can also track important metrics like the LLC miss rate to help identify memory-intensive threads whose performance strongly depends on the memory subsystem (note that a page that is frequently accessed but also hits in the caches does not benefit much from being migrated to fast memory) [26].

Other (software-initiated) methods. BadgerTrap [6] is a kernel extension that intercepts TLB misses and can be used to collect access patterns for selected pages [27]. When a page is chosen for access counting, the kernel (1) *poisons* that page's PTE by setting a reserved bit (bit 51) and (2) flushes the PTE from the TLB. The next access to the page will incur a hardware page walk (due to the TLB miss) and

[2] There are many methods for obtaining memory traces by instrumenting an application (with Pin) and emulating/simulating a system on which the app is running (with QEMU, SimNow, or gem5). But the associated slow-down from these methods makes them more appropriate for postmortem analyses in domains like scientific computing. Such methods are not well suited to fast, on-the-fly memory scheduling of unmodified workloads in a typical data-center environment.

[3] Recently, additional monitoring metrics, such as cache occupancy and memory bandwidth, have been made available via the Resource Control hardware feature [22].

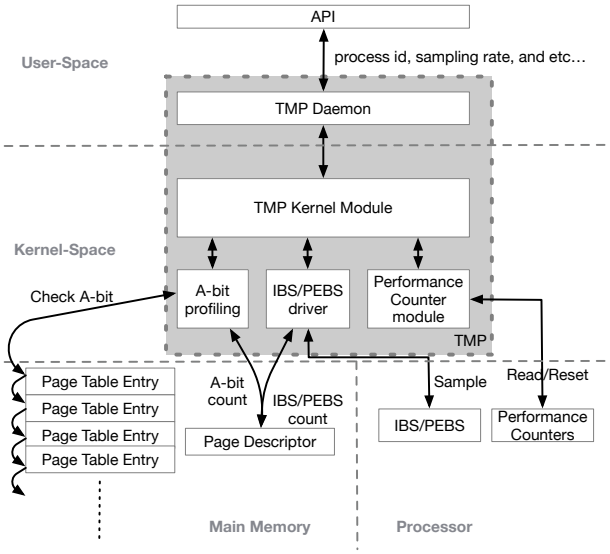


Fig. 1: The architecture of TMP

trigger a protection fault (due to the poisoned PTE), which is intercepted. BadgerTrap's fault handler *unpoisons* the page, installs a valid translation in the TLB, and then *repoisons* the PTE. The total number of BadgerTrap faults thus yields an estimate of the number of TLB misses to a given page. This estimate is then used as a proxy for the number of memory accesses. Such an approach is prone to fault overhead and assumes that the number of TLB misses and the number of cache misses to a page are similar, which may not hold for hot pages, especially of larger sizes.

III. 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.

A. TMP Architecture

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-bit, IBS/PEBS, 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 statistics.

TMP's use of multiple, complementary monitoring methods maximizes informativeness and minimizes overhead (Table I). TMP takes advantage of trace-based profiling methods

Method	Advantages	Disadvantages
Trace-based profiling (IBS/PEBS)	• High accuracy: addresses of individual accesses are supplied. • Overhead is independent of the number of processes tracked.	• Sparse: trace is heavily sampled, so the most accessed pages are noticed. • Raising the sampling rate also increases collection overhead.
PTE A-bit profiling	• Good accuracy: addresses of accessed pages are supplied. • Exact (not sampled): if the page has been accessed, the A bit will eventually be set by PTW: no information is lost due to sampling.	• Overhead is directly proportional to the number of processes tracked: since each PID has a dedicated page table, the more PIDs are covered, the more overhead there is in traversing PTEs.
Performance counters	Stats (e.g., cache misses, bus transactions) can be tracked with close to no overhead.	• If the number of events tracked in parallel exceeds the number of PMU registers, the verbosity suffers due to multiplexing. • Very coarse-grained: one metric for all the instructions running on a core/sharing LLC.

TABLE I: The monitoring methods that TMP employs

(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 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 HWPCs, which incur minimal collection overhead when identifying periods of inactivity.

TMP's profiling mechanisms focus on demand loads because demand loads are on the critical path for the application [28]–[30]; increasing the number of demand loads served from fast memory reduces an application's effective memory-access latency. While prefetcher loads are important on their own, serving them from fast memory won't result in a significant decrease in effective access latency because applications load prefetcher-related data from cache rather than memory (the data having been placed in the cache by the prefetcher ahead of time).

B. TMP Implementation Details

The software TMP prototype includes (a) a kernel driver that interacts with underlying profiling mechanisms and (b) a user-space module that processes collected data and interacts with the application if necessary. This section describes their implementations in detail.

1) IBS/PEBS driver: We extended existing Linux kernel drivers to create an IBS/PEBS driver for TMP [18], [19], [31]. The IBS/PEBS driver uses a hardware mechanism that

receives IBS/PEBS trace samples from underlying microprocessor cores, and the kernel module periodically retrieves the samples. We use machine-specific registers (MSR) to collect memory-trace samples into a kernel buffer and use a register interrupt handler to indicate when the tracing buffer is full. For each access sample, we record the timestamp, CPU id, PID, instruction pointer, virtual data address, physical data address, access type (i.e., whether an access is a load or a store), and useful cache-miss statistics. PD is used to accumulate access counts of both A-bit and IBS/PEBS. *phys_to_page()* returns a pointer to a PD from the physical address.

2) A-bit driver: TMP's A-bit driver uses a software mechanism to directly access PTEs. To visit valid PTEs, the `mm_walk` data structure is used. A callback function registered with `mm_walk` executes when the page walker visits a valid PTE. We register `gather_a_history()` into `mm_walk` for A-bit checking; `gather_a_history()` checks, saves, and clears the A-bit via the `TestClearPageReferenced()` assembler routine. The A-bit driver sees a virtual address and the page table entry the address is mapped to, the driver calls *vm_normal_page()* to obtain a PD.

3) User-space module: We modified `numa_maps` in the Linux `proc` pseudo filesystem to provide a convenient user-space interface for accessing collected profiling information. In user space, TMP requires a profiling daemon to supply PIDs. Whenever the user adds a program to profile, the TMP profiling daemon will signal TMP to collect statistics for all processes forked by the program. TMP can then access these instances' page-table structures via `vm_area_struct`.

4) Optimizations: To reduce collection overhead, TMP incorporates three primary performance optimizations. First, TMP complements A-bit profiling and trace collection with TLB and LLC miss counters, respectively, to dynamically disable profiling when it is not needed. A user-configurable parameter turns each profiling method on and off. In our analysis, we periodically count the number of TLB and LLC misses and update the maximum value counted during a given period. If the current number of events is more than 20% of the maximum, we consider the corresponding profiling method active. The TMP daemon then tells the TMP driver whether to stop or resume A-bit or trace-based profiling.

Second, we filter processes by resource usage (selecting processes with at least 5% CPU or 10% memory) in order to reduce the number of page tables traversed for A-bit collection. We re-evaluate processes once per second. TMP also offers a mode that allows more restrictive filtering to reduce the number of tracked PIDs and keep overhead stable.

Third, we follow prior work [32] and existing kernel routines[4] and do not issue a TLB shootdown after the code clears the A bit of each valid PTE traversed—this reduces the overhead of A-bit collection dramatically because the TLB shootdown results in a costly Inter-Processor Interrupt (IPI) on x86 systems [7], [32], [34], [35]. It is important to note that when an A bit is cleared without a shootdown, the PTE might not reflect a page access for a short period of time. That is, the next A-bit setting might be slightly delayed until the TLB entry is evicted and the page-table walk is triggered (alternatively, a page could be accessed multiple times without the hardware setting the A bit back to 1). We therefore retain a configuration option that allows TMP to issue a shootdown if a software application requires one.

[4]The description of `ptep_clear_flush_young()` in `pgtable.c` states: "On x86 CPUs, clearing the accessed bit without a TLB flush doesn't cause data corruption... So as a performance optimization don't flush the TLB when clearing the accessed bit, it will eventually be flushed by a context switch or a VM operation anyway [35]".

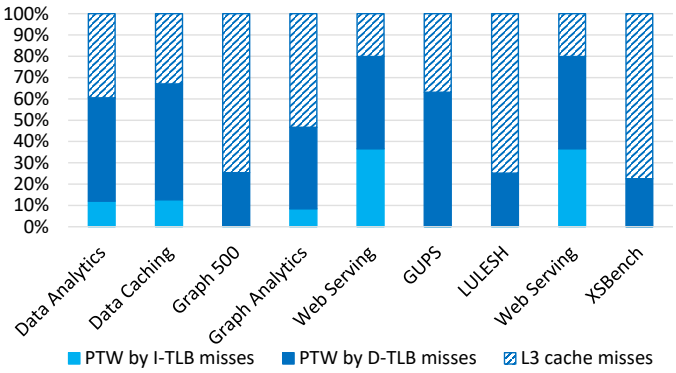


Fig. 2: Ratio of PTW events to cache-miss events.

IV. TMP in Tiered Memory

TMP improves performance of systems with tiered-memory by providing a holistic picture of applications' memory-access behavior. Using TMP's statistics, the system can more efficiently use memory pages located on different memory tiers to improve the fast memory hitrate. We now describe the steps of the TMP-powered tiered memory page placement mechanism.

Step 1: TMP profiler first gathers statistics on workload memory accesses to identify hot and cold pages. TMP then aggregates memory-access statistics for each page from multiple profiling methods into a single hotness rank. The higher the rank, the higher the number of page accesses expected in the future. Pages that rank higher are placed into fast memory because they should benefit the most from tier 1 memory's low latency and/or high bandwidth. By aggregating memory-access statistics from multiple profiling methods into a single hotness rank, TMP offers superior accuracy and performance compared to other profiling implementations.

Figure 2 shows the relative frequency of PTW instruction and data TLB events that set the A bit versus the data-cache miss events tracked using trace-based methods. The number of samples is comparable for the two profiling methods (same order of magnitude), so TMP calculates the rank as a simple sum of A-bit and trace-based samples without the risk of underestimating the impact of either profiling method.

Step 2: Tiered memory policy determines which pages to move across tiers based on page ranks supplied by the tiered profiler (after filtering for non-migratable pages). Researchers have

Policy	Description
Oracle	Assumes knowledge of how many times each page will be accessed in the coming epoch and brings in the hottest (most frequently accessed) pages into the tier 1 memory at the start of the epoch. Oracle represents the upper limit for policy design.
History	A simple yet practical policy that, at the start of each epoch, brings the previous epoch's hottest pages into tier 1 memory.

TABLE II: Tiered-memory policies considered in the study

Name	Input	Type	Configuration
Data-Analytics	Wiki dataset Size: 0.6GB	CloudSuite	1 master 32 workers
Data-Caching	Twitter dataset Size: 36GB	CloudSuite	4 memcached 8 clients
Graph500	Input size:1GB	HPC	8 processes
Graph-Analytics	Twitter dataset Size: 1.4 GB	CloudSuite	1 master 16 workers
GUPS	Input size: 4GB	HPC	8 processes
LULESH	Input size: 21GB	HPC	8 processes
Web-Serving	Faban workload generator	CloudSuite	3 servers 100 clients
XSBench	Input size: 120GB	HPC	8 processes

TABLE III: Workload setup

proposed policy variants implemented in the memory controller, compiler, OS, runtime, hypervisor, or application [2], [32], [36]–[48]. Table II lists the two policies from prior work that we use in our evaluation. The policies are epoch-based; they move pages periodically in batch during designated moments called epoch horizons [2].

We chose the History and Oracle policies described in Table II for the following reasons:
(1) TLB shootdown overhead is prohibitively expensive when moving individual pages because it requires issuing a separate, system-wide shootdown for each page migrated. Epoch-based policies permit a single shootdown at the turn of the epoch for all of the pages moved [2].
(2) Accessing tier 2 memory is not orders of magnitude slower than accessing tier 1 memory; to justify the migration cost, the hottest pages should be migrated. Identifying those pages requires hotness rankings accumulated over a period of time—the epoch duration.

Step 3: The page mover implements policy decisions by migrating the hot pages to tier 1 memory and the cold pages to tier 2 memory. Page migration allows pages to be physically relocated across tiers to the relevant workloads with transparency and while processes are running. So host virtual addresses do not change.

The page-mover design has also been explored in the literature [34], [35], [49]–[54]. Briefly, the page mover is implemented either (a) inside the kernel memory manager with optimizations for page migration (e.g., by making the TLB [illegible]) or (b) on the user level with system calls. The latter case is similar to using the Linux `move_pages()` or `mbind()` with the `MPOL_MF_MOVE` flag and hardware assists from DMA engines (to relieve cores from byte copying).

V. Evaluation Methodology

To evaluate TMP's performance, we developed tools that utilize different profiling methods on the modern processor architectures. We then ran the profiling tools using a set of diverse memory-intensive workloads. This section describes the system configuration and the chosen workloads.

A. Testbed

We ran our experiments on retail CPU models because they are widely available and because their profiling features are common among all models of the same generation.

We used an AMD Ryzen™ 5 3600X CPU with six 3.8 GHz processor cores and a 32 MB LLC. The machine contains 64 GB DRAM and uses a 1 TB SSD to store the workloads and the OS. It runs Linux with kernel version 4.15.18.

B. Workloads

To verify each profiling method, we used memory-intensive workloads (Table III). We selected a subset of CloudSuite and HPC applications, and we chose different inputs and a different number of instances for each workload. To make the profiling effect clearer, we set up each application to use a large memory space. For CloudSuite, we created multiple instances to increase memory usage. For HPC, we have increased the problem size and created multiple processes to increase memory usage.

VI. Experimental Results

To balance the trade-offs between profiling resolution and overhead, TMP must select an appropriate sampling rate for IBS. Through our real-system evaluation, we found sampling one out of every 256K instructions can provide a representative picture of memory behavior while still maintaining a workload overhead that is less than 5% of application overhead. With the aforementioned sampling rate and the corresponding statistics from TMP, the system can significantly improve the end-to-end latency of an application—by as much as 1.13× when using dynamic page allocation. We now discuss these results in detail.

A. Trace-Based Sampling Rate

IBS relies on hardware features to generate statistics about memory accesses but still needs TMP to periodically poll and copy results back to a main memory buffer. To lower the profiling overhead, TMP must find a reasonable sampling frequency. At the same time, TMP must preserve the most important profiling characteristics.

By comparing each workload under different sampling rates, we found that using a sampling rate 4× the default (1 out of 262,144 ops for IBS) better captures the access-pattern details than the default sampling rate does. Table IV shows the number of pages captured by A-bit and IBS profiling with different sampling rates. Compared to the default sampling rate, the 4× rate improves the visibility of memory accesses by 2.58×. In contrast to this improvement, the 8× sampling rate offers less than a 40% improvement over the 4× rate,

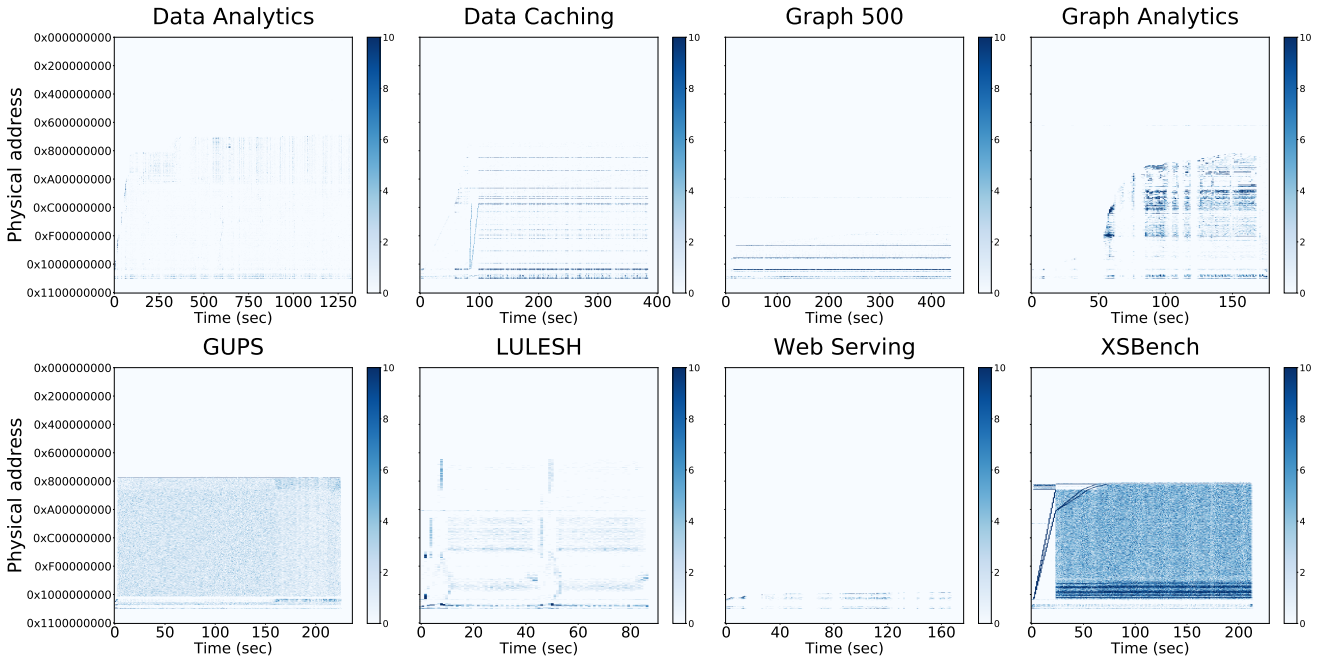


Fig. 3: The heatmap for workload memory accesses using IBS with the 4× sampling rate

	Detected Pages (IBS Default)			Detected Pages (IBS 4x samples)			Detected Pages (IBS 8x samples)		
Workload	A bit	IBS	Both	A bit	IBS	Both	A bit	IBS	Both
Data Analytics	111175	58740	1425	105776	95030	2279	111546	112238	2767
Data Caching	14817	11835	30	14791	15042	49	14768	14586	86
Graph 500	5458	4896	16	5426	9107	42	5423	11548	41
Graph Analytics	28323	28260	104	28051	85161	188	28063	115208	185
GUPS	5587	76009	16	5552	270555	33	5562	468487	36
LULESH	5735	6940	3	5570	20705	19	5584	34134	21
Web Serving	25220	3002	71	25186	4263	109	25199	3610	87
XSBench	5301	199609	8	5284	586787	18	5279	825973	19

TABLE IV: Count of pages captured with two profiling methods. ”Both” shows pages with at least a sample from each method.

and the additional pages that the 8× sampling rate captures are less frequently accessed compared to those captured by the 4× sampling rate. Since the 4× sampling rate has lower sampling overhead, we used it as our sampling rate for IBS in the rest of our experiments.

Figure 3 shows the workload heatmaps for the 4× sampling rate. The horizontal axis of each heatmap shows the time elapsed from when the application started running, and the vertical axis shows the physical memory-address space used. Each temperature point in the heatmap represents the number of times a page frame has been accessed in a given interval.

B. Profiling Comparison

Figure 4 shows the heatmap for each workload with A-bit profiling. A-bit profiling provides complementary information about memory accesses related to the virtual-memory subsystem. Table IV and the cumulative distribution functions (CDFs) for A-bit profiling shown in Figure 5(d) underscore [illegible] to IBS in Figure 5(a)-(c), A-bit profiling alone results in the memory allocator classifying fewer than 15% of the pages that incur TLB misses as hot, so opportunities for performance optimizations are lost.

Table IV lists the number of pages captured by IBS and A-bit profiling for the different workloads and sampling rates. For many data-intensive applications like GUPS and XSBench, IBS classifies 50% more accessed pages as potentially hot. That being said, both A-bit profiling and IBS profiling revealed that the hottest pages constitute a relatively minor portion of the overall memory footprint of many workloads.

To quantify the overhead of each profiling mechanism, we measured the end-to-end latency of each workload with our profiler. For A-bit profiling, we walked through the page table every second. Workload overhead was less than 1% of application overhead. For IBS sampling, we collected memory-trace samples with the default and 4× sampling rates. In all cases, the workload overhead was under 5% of application overhead, and even lower (under 2%) for the default sampling rate.

C. Performance Gains with Tiered-Memory Placement

We first analyze the tiered-memory policies from Section IV. Figure 6 measures the memory hitrate—the number of tier 1 memory accesses relative to the total number of accesses to both tiers (a key metric for any TMA system). The results are based on the profiling data from the real hardware. We

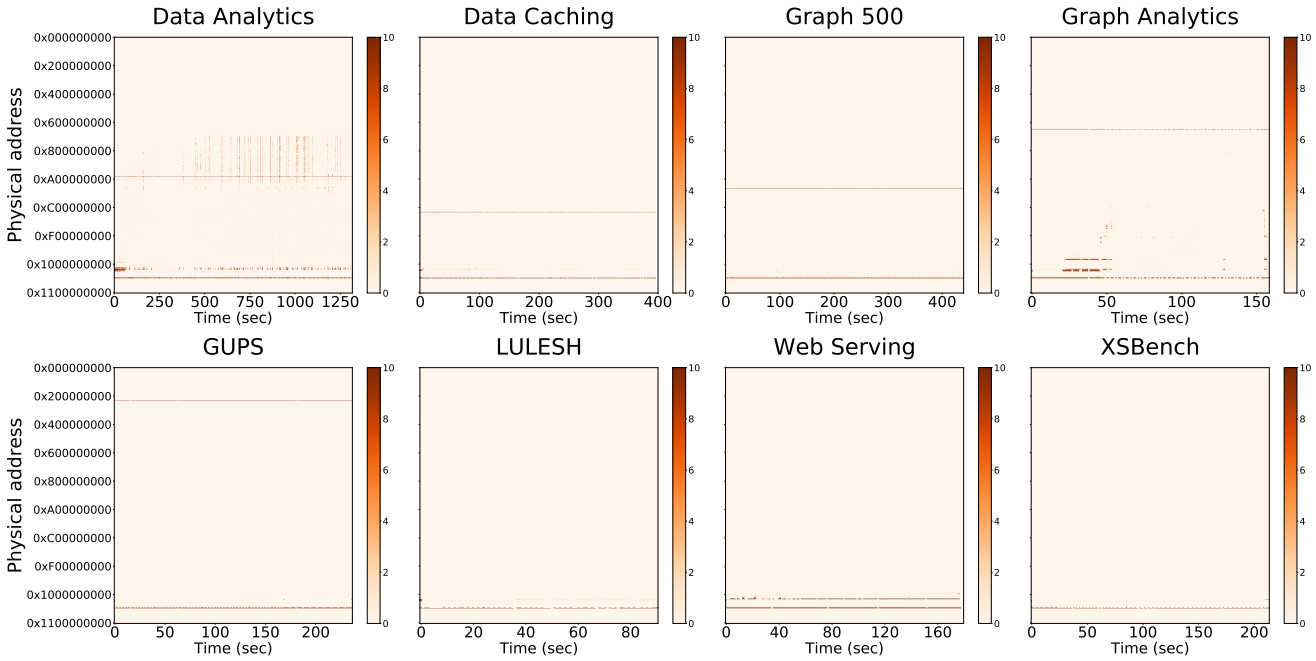
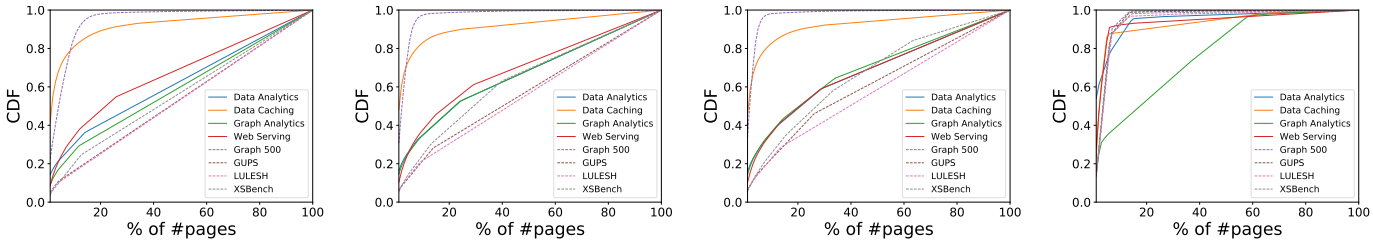


Fig. 4: The heatmap for workload memory accesses using A-bit profiling



(a) default sampling rate (b) 4× sampling rate (c) 8× sampling rate (d) A-bit

Fig. 5: The CDFs for memory accesses using different workloads, profiling techniques, and sampling rates

include ratios ranging from 1/8 to 1/128; the smaller the ratio, the greater the challenge for a policy to find the right pages to place into (ever-shrinking) tier 1 memory.

We calculated the policy results using (a) A-bit profiling alone, (b) IBS trace-based profiling alone, and (c) both A-bit and IBS monitoring data from TMP. Figure 6 shows that the performance of the Oracle policy is better on the combined data versus the "piecemeal" monitoring approach, often by as high as 70%. The simple, reactive History policy often lags behind the predictive Oracle policy due to complicated access patterns and the fact that many applications exhibit Monte Carlo or randomized accesses. Unlike the Oracle policy, the History policy sometimes struggles to properly combine the data from multiple monitoring sources. Yet we see that even the History policy often outperforms the non-TMP monitoring methods by as much as 60%.

To evaluate the effects of improved memory hit rates from [illegible], we developed an emulation framework based on BadgerTrap [6]. We chose emulation because available NVM media (e.g., Intel's 3D XPoint memory [55]) can work with a limited combination of processor architectures, special motherboards, and BIOS support. NVM-enabled systems also require kernel hacking (as opposed to application development for persistent memory), which makes them difficult to use remotely, even in a virtualized setting.

In our emulation testbed, we maintain a list of slower memory locations and set protection bits on memory pages that belong to the list. When an attempt is made to reach one of these protected pages, the trap handler adds latency before the system can grant access to the page. The emulation framework sets the protection bits periodically, thus inserting additional latency to emulate slower memory. We used the latency of modern non-volatile main-memory technologies to calibrate our timing parameters for the emulation framework. We use 50 us as the page migration overhead, 10 us for each slow memory access after a protection fault, and an additional 13 us latency if the page in the slow memory is hot. We configured the system with 4 GB of fast tier 1 memory combined with 60 GB of slower tier 2 memory. The results show that TMP achieves an average speedup of 1.04×, and an optimal speedup of 1.13×, compared with the baseline that uses a NUMA-like, first-come-first-allocate, tiered-memory policy.

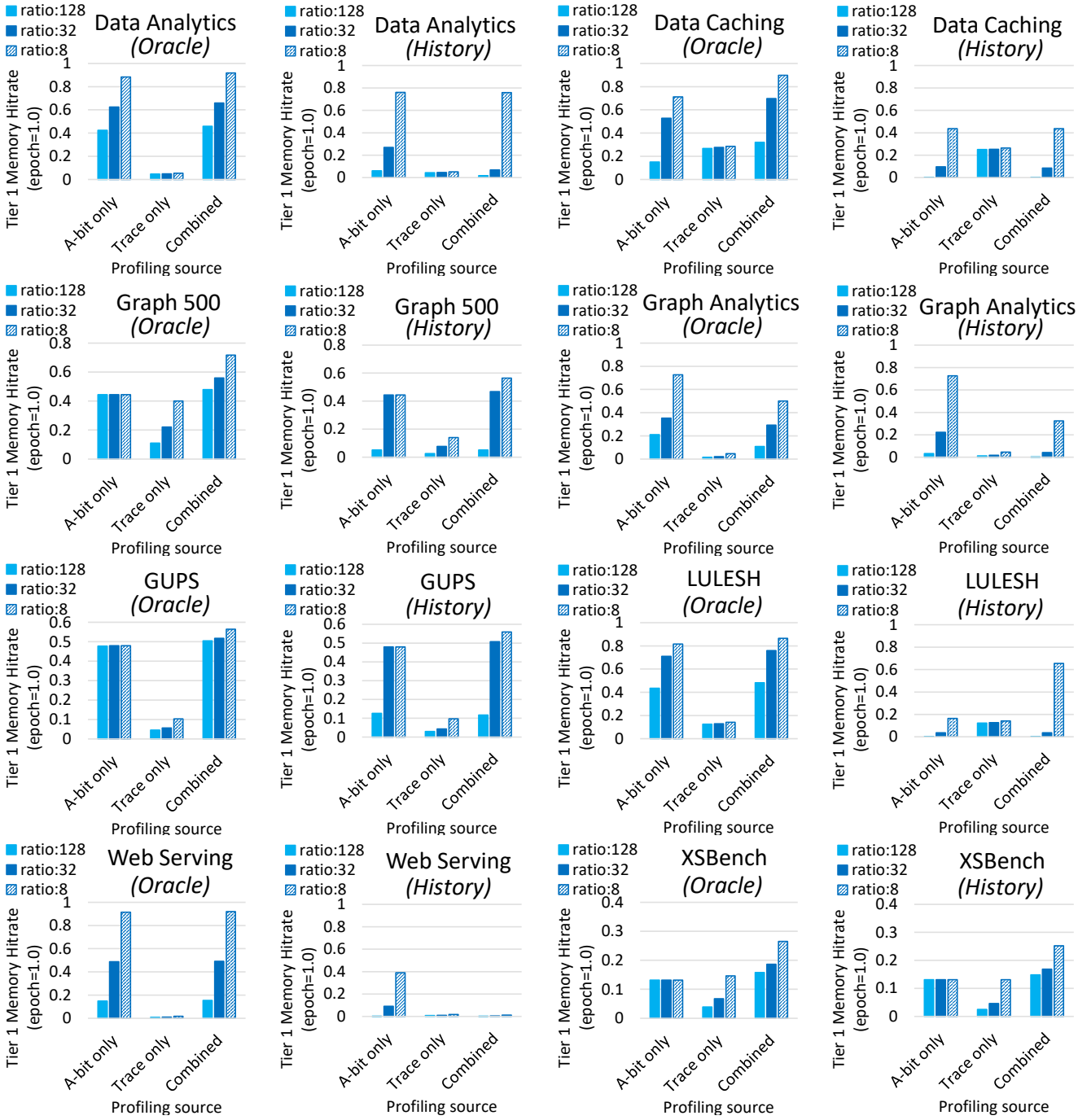


Fig. 6: The hitrate in the first tier of memory for the Oracle and History policies with an epoch of 1 second

VII. Related Work

There is a limited body of work on memory profiling with NUMA and/or tiered-memory. MemBrain [46] uses counters for an offline estimation of memory-bandwidth utilization. MemBrain also measures the overhead of LLC miss-trace collection with PEBS-based profiling. The offline profiling [6] previously showed that long-latency loads provide a better indicator for page placement than TLB misses. Thermostat [27] classifies pages as hot or cold by intercepting TLB misses via BadgerTrap [6], and Carrefour [53] combines counters with IBS profiling to manage threads and memory to avoid traffic hotspots, thereby preventing congestion in memory controllers and on NUMA links. These implementations do not target a combined, dynamic, vendor-agnostic profiling interface, which is the point of our work.

Some of the biggest supercomputers are expected to employ tiered memory in the near future [57]. Runtime projects Simplified Interface to Complex Memory [58] and Umpire [59] seek to help port existing HPC codes to the new memory paradigm. Our system is complementary to these projects.

VIII. Conclusion

Tiered memory promises exabytes of byte-addressable memory for software. The challenge is that software does not see memory accesses by default. By leveraging multiple HW-profiling methods available on modern CPUs, we argue for a comprehensive, hybrid solution that makes the memory-access stream visible to software. This visibility can then be leveraged to achieve informed tiered-memory placement.

The authors would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by two awards, CNS-1940048 and CNS-2007124, from National Science Foundation (NSF).

References

[1] J. Handy and the Linley Group, "Emerging memories: Why now? opportunity opens for 3D XPoint, MRAM, and similar designs," 2019.
[2] 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*.
[3] D. Magenheimer, C. Mason, D. McCracken, K. Hackel, and O. Corp, "Transcendent Memory and Linux," 2006.
[4] 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.
[5] https://www.kernel.org/doc/html/latest/vm/hmm.html.
[6] 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.
[7] M. Oskin and G. H. Loh, "A software-managed approach to die-stacked DRAM," PACT 2015, pp. 188–200.
[8] http://linux.die.net/man/3/numa.
[9] https://www.kernel.org/doc/html/latest/vm/page_migration.html.
[10] "Persistent memory for transient data." https://lwn.net/Articles/777212/.
[11] "NUMA nodes for managing PMEM." https://lwn.net/Articles/787418/.
[12] "How to Use PMEM as NUMA node." https://lwn.net/Articles/783811/.
[13] https://github.com/HewlettPackard/LinuxKI/wiki/NUMA-Balancing.
[14] G. H. Loh and M. D. Hill, "Supporting Very Large Caches with Conventional Block Sizes," in *MICRO 2011*.
[15] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers," in *Proc. of the Intl. Symp. on Computer Architecture*, 2013.
[16] A. Bhattacharjee, D. Lustig, and M. Martonosi, *Architectural and Operating System Support for Virtual Memory*. 2017.
[17] https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/page-modification-logging-vmm-white-paper.pdf.
[18] https://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf.
[19] https://github.com/jlgreathouse/AMD_IBS_Toolkit.
[20] AMD, "AMD64 technology: Lightweight profiling specification," 2010.
[21] "Intel 64 and IA-32 architectures software developer's manual, vol. 3."
[22] "User interface for resource control feature." https://www.kernel.org/doc/html/latest/x86/resctrl_ui.html.
[23] S. Eranian, "What can performance counters do for memory subsystem analysis?," in *2008 MSPC*.
[24] *http://linuxplumbersconf.org/2009/slides/Arnaldo-Carvalho-de-Melo-[illegible].pdf*.
[25] [illegible] "Advanced hardware profiling and sampling (PEBS, IBS, etc.): Creating a new PAPI sampling interface," 2016.
[26] [illegible] Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS Observations to Improve Performance in Multicore Systems," *IEEE Micro*, vol. 28, no. 3, 2008.
[27] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," ASPLOS 2017.
[28] E. S. Tune, D. M. Tullsen, and B. Calder, "Quantifying instruction criticality," in *PACT 2002*, pp. 104–113.
[29] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, "Using interaction costs for microarchitectural bottleneck analysis," in *MICRO-36*.
[30] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh, "Criticality-based optimizations for efficient load processing," in *HPCA 2009*.
[31] "`perf` Examples." http://www.brendangregg.com/perf.html.
[32] S. Lee, H. Bahn, and S. H. Noh, "CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures.," *IEEE Trans. Computers*, pp. 2187–2200, 2014.
[33] https://patchwork.kernel.org/patch/10631917/.
[34] M. K. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy translation coherence," ASPLOS 2018, pp. 651–664.
[35] N. Amit, "Optimizing the TLB shootdown algorithm with page access tracking," in *USENIX ATC 17*, pp. 27–39.
[36] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter," in *ISCA 2017*.
[37] V. Gupta, M. Lee, and K. Schwan, "HeteroVisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms," VEE 2015.
[38] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A hybrid memory page scheduler with machine intelligence," HPDC 2019, pp. 37–48, 2019.
[39] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *ICS 2011*.
[40] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," EuroSys 2016, pp. 15:1–15:16.
[41] D. Shen, X. Liu, and F. X. Lin, "Characterizing emerging heterogeneous memory," ISMM 2016, pp. 13–23.
[42] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime data management on non-volatile memory-based heterogeneous main memory," SC 2017.
[43] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," SC 2018, pp. 31:1–31:13.
[44] W. Zhang and T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures," in *PACT 2009*.
[45] T. D. Doudali and A. Gavrilovska, "CoMerge: Toward efficient data placement in shared heterogeneous memory systems," MEMSYS 2017.
[46] M. B. Olson, B. Kammerdiener, M. R. Jantz, K. A. Doshi, and T. Jones, "Portable application guidance for complex memory systems," *MEMSYS*, 2019.
[47] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new OS architecture for scalable multicore systems," SOSP 2009.
[48] I. B. Peng, S. Markidis, E. Laure, G. Kestor, and R. Gioiosa, "Exploring application performance on emerging hybrid-memory supercomputers," *2016 HPCC/SmartCity/DSS*, 2016.
[49] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," ASPLOS 2019, pp. 331–345.
[50] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," *SIGARCH CompArch News*, vol. 44.
[51] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris, "Shoal: Smart allocation and replication of memory for parallel programs," in *USENIX ATC 15*, pp. 263–276.
[52] https://patchwork.kernel.org/cover/10743133/.
[53] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on NUMA systems," ASPLOS 2013, pp. 381–394.
[54] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on NUMA systems: Asymmetry matters," USENIX ATC 2015.
[55] https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.
[56] J. Marathe and F. Mueller, "Hardware profile-guided automatic page placement for CcNUMA systems," PPoPP 2006.
[57] https://www.energy.gov/articles/us-department-energy-and-intel-build-first-exascale-supercomputer.
[58] M. Lang, L. Ionkov, and S. Williams, "Simplified interface to complex memory hierarchies," 2017.
[59] D. A. Beckingsale and R. D. Hornung, "Umpire: Status report," 2018.