Thermostat: Application-transparent Page Management for Two-tiered Main Memory

Neha Agarwal Thomas F. Wenisch
University of Michigan
nehaag@umich.edu, twenisch@umich.edu

Abstract

The advent of new memory technologies that are denser and cheaper than commodity DRAM has renewed interest in two-tiered main memory schemes. Infrequently accessed application data can be stored in such memories to achieve significant memory cost savings. Past research on two-tiered main memory has assumed a 4KB page size. However, 2MB huge pages are performance critical in cloud applications with large memory footprints, especially in virtualized cloud environments, where nested paging drastically increases the cost of 4KB page management. We present *Thermostat*, an application-transparent huge-page-aware mechanism to place pages in a dual-technology hybrid memory system while achieving both the cost advantages of two-tiered memory and performance advantages of transparent huge pages. We present an online page classification mechanism that accurately classifies both 4KB and 2MB pages as hot or cold while incurring no observable performance overhead across several representative cloud applications. We implement Thermostat in Linux kernel version 4.5 and evaluate its effectiveness on representative cloud computing workloads running under KVM virtualization. We emulate slow memory with performance characteristics approximating near-future high-density memory technology and show that Thermostat migrates up to 50% of application footprint to slow memory while limiting performance degradation to 3%, thereby reducing memory cost up to 30%.

CCS Concepts **• Software and its engineering → Memory management**

Keywords Cloud Computing, Operating systems

ASPLOS '17 April 8–12, 2017, Xi'an, China.

DOI: http://dx.doi.org/10.1145/3037697.3037706

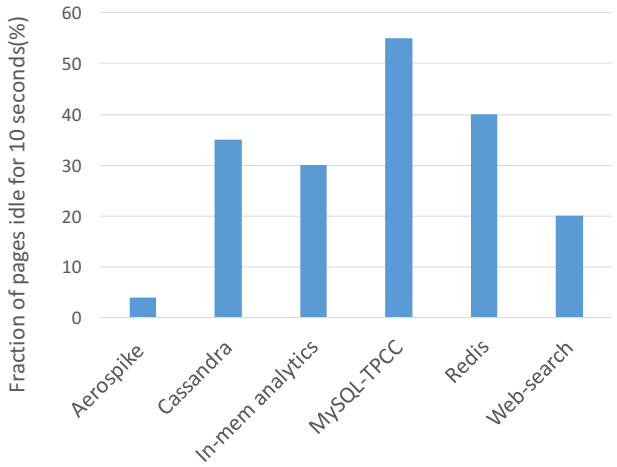

Figure 1. Fraction of 2MB pages idle/cold for 10s detected via per-page *Accessed* bits in hardware. Note that this technique cannot estimate the page access rate, and thus cannot estimate the performance degradation caused by placing these pages in slow memory (which exceeds 10% for Redis).

1. Introduction

Upcoming memory technologies, such as Intel/Micron's recently-announced 3D XPoint memory [29], are projected to be denser and cheaper per bit than DRAM while providing the byte-addressable load-store interface of conventional main memory. Improved capacity and cost per bit come at the price of higher access latency, projected to fall somewhere in the range of 400ns to several microseconds [29] as opposed to 50–100ns for DRAM. The impending commercial availability of such devices has renewed interest in two-tiered physical memory, wherein part of a system's physical address space is implemented with the slower, cheaper memory technology [22, 41].

Slow memory can result in a net cost win if the cost savings of replaced DRAM outweigh cost increase due to reduced program performance or by enabling a higher peak memory capacity per server than is economically viable with DRAM alone. To realize cost savings, in this study, we set

an objective of at most 3% performance degradation relative to a DRAM-only system.

However, making effective, transparent use of slow memory to reduce cost without substantial performance loss is challenging. Any memory placement policy must estimate the performance degradation associated with placing a given memory page in slow memory, which in turn requires some method to gauge the *page access rate*. Lack of accurate page access rate tracking in contemporary x86 hardware makes this task challenging. Moreover, as we will show, naive policies to place pages into slow memory based on existing hardware-maintained *Accessed* bits are insufficient and can lead to severe performance degradations.

Making slow memory usage application transparent is particularly critical for cloud computing environments, where the cloud provider may wish to transparently substitute cheap memory for DRAM to reduce provisioning cost, but has limited visibility and control of customer applications. Relatively few cloud customers are likely to take advantage of cheaper-but-slower memory technology if they must redesign their applications to explicitly allocate and manage hot and cold memory footprints. A host-OS-based cold memory detection and placement mechanism is a natural candidate for such a system. Figure 1 shows the amount of data idle for 10s as detected at runtime by an existing Linux mechanism to monitor hardware-managed *Accessed* bits in the page tables for various cloud applications. We observe that substantial cold data (more than 50% for MySQL) can be detected by application-transparent mechanisms.

However, there has been little work on providing performance degradation guarantees in the presence of page migration to slow memory [33]. Furthermore, prior work on two-tiered memory has assumed migration/paging at 4KB page granularity [22, 41]. However, huge pages (2MB pages) are now ubiquitous and critical, especially for cloud platforms where virtualization magnifies the costs of managing 4KB pages. We observe performance improvements as high as 30% from huge pages under virtualization (Table 1). Our proposal, *Thermostat*, manages two-tiered main memory transparently to applications while preserving the benefits of huge pages and dynamically enforcing limits on performance degradation (e.g., limiting slowdown to 3%). We will show that Thermostat is huge-page-aware and can place/migrate 4KB and huge pages while limiting performance degradation within a target bound.

Prior work has considered two approaches to two-tiered memory: (i) a paging mechanism [34, 35], wherein accesses to slow memory invoke a page fault that must transfer data to fast memory before an access may proceed, and (ii) via a migration mechanism (as in cache coherent NUMA multiprocessors) [18], wherein no software fault is required. In the latter scenario, a migration mechanism seeks to shuffle pages between tiers to maximize fast-memory accesses. Dulloor et al. [22] have described a programmer-guided data placement scheme in NVRAM. Such techniques are inapplicable when cloud customers run third-party software and do not have access to source code. Li et al. [33] describe a hardware-based technique to accurately gauge the impact of moving a page from NVRAM to DRAM. However, such hardware requires significant changes to contemporary x86 architecture. In contrast, Thermostat does not require *any* additional hardware support apart from the availability of slow memory.

	Performance gain
Aerospike	6%
Cassandra	13%
In-memory Analytics	8%
MySQL-TPCC	8%
Redis	30%
Web-search	No difference

Table 1. Throughput gain from 2MB huge pages under virtualization relative to 4KB pages on both host and guest.

To provide a low overhead cold-page detection mechanism, Thermostat continuously samples a small fraction of pages and estimates page access rate by spatial extrapolation (described in Section 3.2). This strategy makes Thermostat both low overhead and fast-reacting. The single *Accessed* bit per page provided by hardware is insufficient to distinguish hot and cold pages with sufficiently low overhead. Instead, Thermostat uses TLB misses as a proxy for LLC misses as they can be tracked in the OS through reserved bits in the PTE, allowing page access rates to be estimated at low overhead. Finally, Thermostat employs a correction mechanism that rapidly detects and corrects mis-classified cold pages (e.g., due to time-changing access patterns).

We implement Thermostat in Linux kernel version 4.5 and evaluate its effectiveness on representative cloud computing workloads running under KVM virtualization. As the cheaper memory technologies we target, such as 3D XPoint, are not yet commercially available, our evaluation emulates slow memory using a software technique that triggers translation faults for slow memory pages, yielding a 1us average access latency.

In summary, we make the following contributions:

- We propose an online low-overhead mechanism for estimating the performance degradation due to placing a particular page in slow memory.
- We use this mechanism in an online, huge-page-aware hot/cold page classification system that *only* requires a target maximum slowdown as input.
- We propose an online method to detect mis-classifications and rectify them, thereby minimizing the impact of such mis-classifications on application throughput.
- By emulating slow memory in software, we demonstrate that Thermostat can migrate up to 50% of cloud appli-

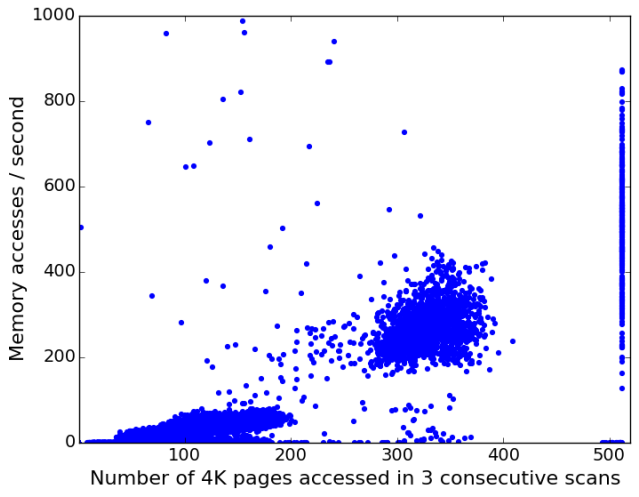

Figure 2. Memory access rate vs. hardware *Accessed* bit distribution of 4KB regions within 2MB pages for Redis. The single hardware *Accessed* bit per page does not correlate with memory access rate per page, and thus cannot estimate page access rates with low overhead.

cation footprints to slow memory with a 3% slowdown, reducing memory provisioning cost up to 30%.

2. Background and Motivation

We briefly motivate the potential for dual-technology main memory and the importance of huge pages under virtualized execution.

2.1 Shifting cold data to cheap memory

For performance-sensitive and high-footprint cloud applications, it is unlikely that cheaper-but-slower memory technologies, such as Intel's 3D XPoint memory [29], will entirely supplant DRAM main memory. An increase in memory access latency by even a small multiple (e.g., to 500ns) will result in drastic throughput losses, as the working set of these applications typically greatly exceeds cache capacities. Because DRAM accounts for only a fraction of total system cost, the net cost of the throughput loss will greatly outweigh the savings in main memory cost.

However, cloud applications typically have highly skewed access distributions, where a significant fraction of the application's footprint is infrequently accessed [17]. These rarely accessed pages can be mapped to slow memory without significantly impacting performance. We refer to this infrequently accessed data as "cold" data.

To distinguish cold pages from frequently accessed hot pages, existing mechanisms exploit the *Accessed* bit in the PTE (set by the hardware each time the PTE is accessed by a page walk) [28, 32, 42]. We investigated one such existing cold-page detection framework, `kstaled` [32]. However, we find that the single accessed bit per page is insufficient to distinguish hot and cold 2MB huge pages with sufficiently low overhead. To detect a page access, `kstaled` must clear the *Accessed* bit and flush the corresponding TLB entry. However, distinguishing hot and cold pages requires monitoring (and hence, clearing) accessed bits at high frequency, resulting in unacceptable slowdowns.

Figure 2 illustrates why hot and cold 2MB huge pages cannot be distinguished by temporally sampling the hardware-maintained access bits at the highest possible rate that can meet our tight performance degradation target (3% slowdown), using Redis as an example workload. The hardware can facilitate monitoring at 4KB granularity by temporarily splitting a huge page and monitoring the accessed bits of the 512 constituent 4KB pages (monitoring at 2MB granularity without splitting provides even worse hot/cold differentiation [28]). The horizontal axis represents the number of detected "hot" 4KB regions in a given 2MB page when monitoring at the maximum frequency that meets our slowdown target. Here "hot" refers to pages that were accessed in three consecutive scan intervals. The vertical axis represents the ground-truth memory access rate for each 2MB page. (We describe our methodology for measuring memory access rate in Section 3.3). The key take-away is that the scatter plot is highly dispersed—the spatial frequency of accesses within a 2MB page is poorly correlated with its true access rate. Conversely, performance constraints preclude monitoring at higher temporal frequency. Hence, mechanisms that rely solely on *Accessed* bit scanning cannot identify cold pages with low overhead.

2.2 Benefits of transparent huge pages

Intel's IA-64 x86 architecture mandates a 4-level page table structure for 4KB memory pages. So, a TLB miss may incur up to four memory accesses to walk the page table. Under virtualization, with two-dimensional page table walks (implemented in Intel's Extended Page Tables and AMD's Nested Page Tables), the cost of a page walk can be as high as 24 memory accesses [11, 30]. When memory is mapped to a 2MB huge page in both the guest and host, the worst-case page walk is reduced to 15 accesses, which significantly lowers the performance overhead of virtualization. Moreover, 2MB huge pages increase the TLB reach and improve the cacheability of intermediate levels of the page tables, as fewer total entries compete for cache capacity.

Table 1 shows the performance benefit of using huge pages via Linux's Transparent Huge Page (THP) mechanism. We compare throughput of various cloud applications where both the guest and host employ 2MB huge pages against configurations with transparent huge pages disabled (i.e., all pages are 4KB). We observe significant throughput benefits as high as 30% for Redis. Previous literature has also reported performance benefits of huge pages [15, 27, 28]. From these results, it is clear that huge pages are essential to performance, and any attempt to employ a dual-technology main memory must preserve the performance ad-

vantages of huge pages. For this reason, we only evaluate Thermostat with THP enabled at both host and guest.

3. Thermostat

We present Thermostat, an application-transparent huge-page-aware mechanism to detect cold pages during execution. The input to Thermostat is a user-specified tolerable slowdown (3% in our evaluation) incurred as a result of Thermostat's monitoring and due to accesses to data shifted to slow memory. Thermostat periodically samples a fraction of the application footprint and uses a page poisoning technique to estimate the access rate to each page with tightly controlled overhead. The estimated page access rate is then used to select a set of pages to place in cold memory, such that their aggregate access rate will not result in slowdown exceeding the target degradation. These cold pages are then continually monitored to detect and rapidly correct any misclassifications or behavior changes. In the following sections, we describe the Thermostat in more detail.

3.1 Overview

We implement Thermostat in the Linux 4.5 kernel. Thermostat can be controlled at runtime via the Linux memory control group (cgroup) mechanism [38]. All processes in the same cgroup share Thermostat parameters, such as the sampling period and maximum tolerable slowdown.

The Thermostat mechanism comprises four components: (i) a sampling mechanism that randomly selects a subset of pages for access-rate monitoring, (ii) a monitoring mechanism that counts accesses to sampled pages while limiting maximum overhead, (iii) a classification policy to select pages to place in slow memory, and (iv) a mechanism to monitor and detect mis-classified pages or behavior changes and migrate pages back to conventional (fast) memory.

The key challenge that Thermostat must address is the difficulty of discerning the access rates of 2MB pages at a sufficiently low overhead while still responding rapidly to changing workload behaviors. Tracking the access rate of a page is a potentially expensive operation if the page is accessed frequently. Hence, to bound the performance impact of access rate monitoring, only a small fraction of the application footprint may be monitored at any time. However, sampling only a small fraction of the application footprint leads to a policy that adapts only slowly to changes in memory behavior.

3.2 Page sampling

Sampling a large number of huge pages is desirable as it leads to quick response to time-varying workload access patterns. But, it can lead to a high performance overhead, since, as explained in Section 3.3, each TLB miss to a sampled page incurs additional latency for OS fault handling. To tightly control application performance slowdown, *we split a random sample of huge pages (5% in our case) into 4KB pages, and poison only a fraction of these 4KB pages in each sampling interval*. Below, we detail the strategy used to select which 4KB pages to poison, and how we estimate the total access rate from the access rates of the sample.

A simple strategy to select 4KB pages from a set of huge pages is to select K random 4KB pages, for some fixed K ($K = 50$ in our evaluation). However, when only a few 4KB pages in a huge page are hot, this naive strategy may fail to sample them, and thus deem the overall huge page to have a low access rate. To address this shortcoming, our mechanism monitors page access rates in two steps. We first rely on the hardware-maintained *Accessed* bits to monitor all 512 4KB pages and identify those with a non-zero access rate. We then monitor a sample of these pages using our more costly software mechanism to accurately estimate the aggregate access rate of the 2MB page. With our strategy, *only 0.5% of memory is sampled at any time*, which makes the performance overhead due to sampling $< 1\%$.

To compute the aggregate access rate at 2MB granularity from the access rates of the sampled 4KB pages, we scale the observed access rate in the sample by the total number of 4KB pages that were marked as accessed. The monitored 4KB pages comprise a random sample of accessed pages, while the remaining pages have a negligible access rate.

3.3 Page access counting

Current x86 hardware does not support access counting at a per-page granularity. Thus, we design a software-only solution to track page access rates with very low overhead ($<1\%$) by utilizing PTE reserved bits. In Section 6.1, we discuss two extensions to existing x86 mechanisms that might enable lower overhead page-access counting.

To approximate the number of accesses to a page, we use BadgerTrap, a kernel extension for intercepting TLB misses [26]. When a page is sampled for access counting, Thermostat poisons its PTE by setting a reserved bit (bit 51), and then flushes the PTE from the TLB. The next access to the page will incur a hardware page walk (due to the TLB miss) and then trigger a protection fault (due to the poisoned PTE), which is intercepted by BadgerTrap. BadgerTrap's fault handler unpoisons the page, installs a valid translation in the TLB, and then repoisons the PTE. By counting the number of BadgerTrap faults, we can estimate the number of TLB misses to the page, which we use as a proxy for the number of memory accesses.

Note that our approach assumes that the number of TLB misses and cache misses to a 4KB page are similar. For hot pages, this assertion does not hold. However, Thermostat has no need to accurately estimate the access rate to hot pages; it is sufficient to know that they are hot. Conversely, for cold pages, nearly all accesses incur both TLB and cache misses as there is no temporal locality for such accesses, and, therefore, tracking TLB misses is sufficient to estimate the page access rate. We validated our approach by measuring the TLB miss rates (resulting in page-walks) and last-level

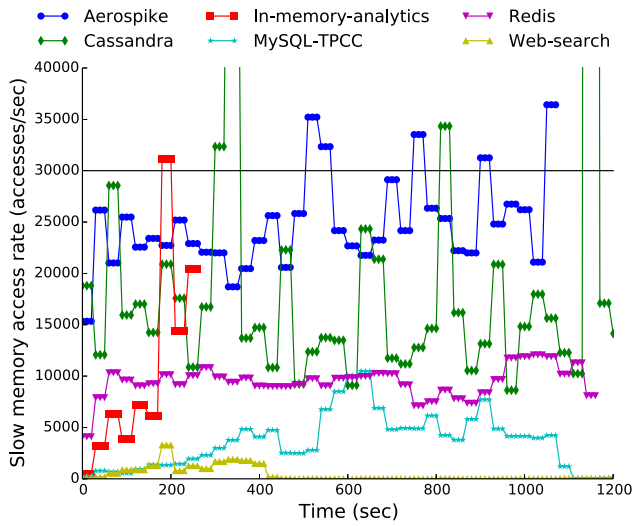

Figure 3. Slow memory access rate over time. For a 3% tolerable slowdown and 1us slow memory access latency, the target slow memory access rate is 30K accesses/sec. Thermostat tracks this 30K accesses/sec target. Note that different benchmarks have different run times; we plot the x-axis for 1200 seconds.

cache miss rates for our benchmark suite using hardware performance counters via the Linux `perf` utility. For pages we identify as cold, the TLB miss rate is typically higher (but always within a factor of two) of the last-level cache miss rate measured without BadgerTrap, indicating that our approach is reasonable.

3.4 Page classification

Classifying pages as hot or cold is governed by the user-specified maximum tolerable slowdown (without such a threshold, one can simply declare all pages cold and call it a day). To select cold pages, we use the estimated access rates of each (huge) page.

We translate a tolerable slowdown of $x\%$ to an access rate threshold in the following way. Given A accesses to slow memory in one second, the total time consumed by slow memory accesses is At_s, where t_s is the access latency of the slow memory. Thus, a slowdown threshold of $x\%$ can be translated to an access rate threshold of $\frac{x}{100t_s}$ per second. If a fraction f of the total huge pages were sampled, we assign pages to the slow memory such that their aggregate estimated access rate does not exceed $f\frac{x}{100t_s}$. We sort the sampled huge pages in increasing order of their estimated access rates, and then place the coldest pages in slow memory until the total access rate reaches the target threshold.

This simple approach governs the access rate to slow memory to avoid the user-specified degradation target. In Figure 3, we show slow memory access rate averaged over 30 seconds for our benchmark suite, assuming 1us slow memory access latency and 3% tolerable slowdown (we discuss detailed methodology in Section 4). We observe that Thermostat keeps the slow memory access rate close to the target 30K accesses/sec. For Aerospike and Cassandra slow memory access rate temporarily exceeds 30K accesses/sec but is brought back below 30K accesses/sec by mis-classification detection, discussed next in Section 3.5.

3.5 Correction of mis-classified pages

Since we estimate the access rate of a huge page based on the access rates of only a few (not more than 50, as described in Section 3.2) 4KB pages, there is always some probability that some hot huge pages will be mis-classified as cold due to sampling error. Such mis-classifications are detrimental to application performance, since the interval between successive samplings of any given huge page can be fairly long. To address this issue, we track the number of accesses being made to each cold huge page, using the software mechanism mentioned in Section 3.3. Since the access rate to these pages is slow by design, the performance impact of this monitoring is low. In every sampling period we sort the huge pages in slow memory by their access counts and their aggregate access count is compared to the target access rate to slow memory. The most frequently accessed pages are then migrated back to fast memory until the access rate to the remaining cold pages is below the threshold. In addition to any mis-classified pages, this mechanism also identifies pages that become hot over time, adapting to changes in the application's hot working set.

3.6 Migration of cold pages to slow memory

Once cold pages have been identified by the guest, they must be migrated to the slow memory. We use the NUMA support in KVM guests to achieve this transfer. The NVM memory space is exposed to the guest OS as a separate NUMA zone, to which the guest OS can then transfer memory. NUMA support in KVM guests already exists in Linux and can be used via `libvirt` [3].

3.7 Thermostat example

Figure 4 illustrates Thermostat's page classification for an example application with eight huge pages. Each sampling period comprises three stages: (i) split a fraction of huge pages, (ii) poison a fraction of split and accessed 4KB pages, record the access count to 4KB pages to estimate access rate of the huge pages, and (iii) classify pages as hot/cold. In this example, we sample 25% of huge pages (two huge pages out of eight are sampled). In the first sampling period, Thermostat splits and records 4KB-grain accesses to two pages (page 1 and 5) in the first scan period. In the second scan period, 4KB pages 1 and 4 of the first huge page and 3 and 8 of the fifth huge page are selected to be poisoned. Thermostat then estimates the access rate to huge page 1 and 5 from the access rates of the 4KB pages. Finally, Thermostat sorts the estimated access rates of the huge pages and classifies page 1 as cold, as its estimated access rate is below

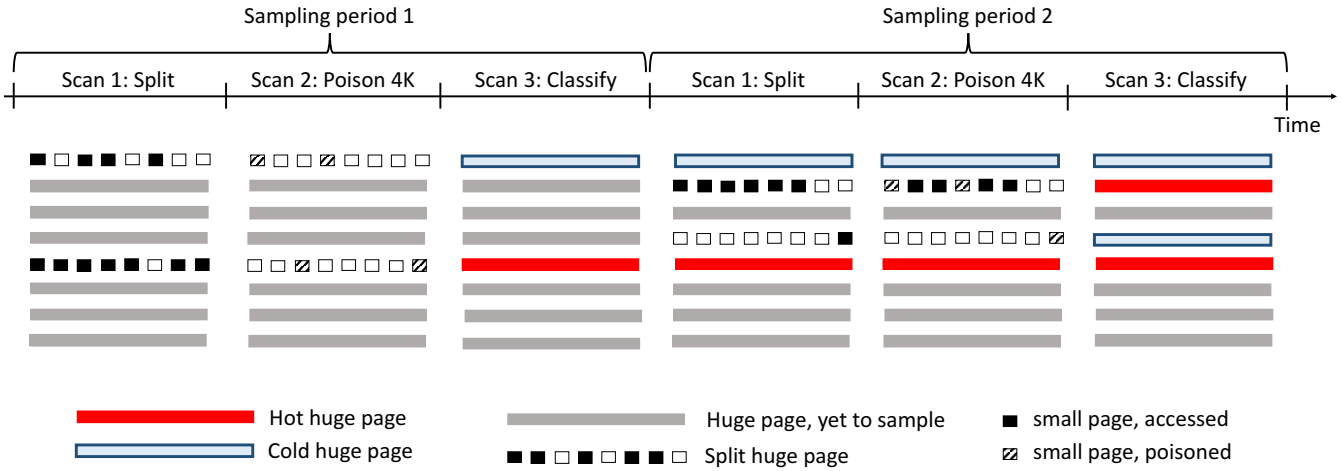

Figure 4. Thermostat operation: Thermostat classifies huge pages into hot and cold by randomly splitting a fraction of pages and estimating huge page access rate by poisoning a fraction of 4K pages within a huge page. The sampling policy is described in Section 3.2, Section 3.3 describes our approach to page access counting, and Section 3.4 describes the page classification policy. Note that the sampling and poisoning fractions here are for illustration purposes only. In our evaluation we sample 5% of huge pages and poison at most 50 4KB pages from a sampled huge page.

the threshold tolerable slow memory access rate. However, because the sum of the access rates of both huge pages is above the threshold access rate, page 5 is classified as hot. Similarly, in the second sampling period, pages 2 and 4 are randomly selected for sampling. At the end of the sampling period page 2 is classified as hot and page 4 as cold.

4. Methodology

4.1 System Configuration

We study Thermostat on a 36-core (72 hardware thread) dual-socket x86 server, Intel Xeon E5-2699 v3, with 512 GB RAM running Linux 4.5. Each socket has 45MB LLC. There is a 64-entry TLB per core and a shared 1024 entry L2 TLB. Several of our benchmark applications perform frequent I/O and are highly sensitive to OS page cache performance. To improve the page cache, we install `hugetmpfs`, a mechanism that enables use of huge pages for the `tmpfs` [19] filesystem. We place all files accessed by the benchmarks in `tmpfs`. In the future, we expect that Linux may natively support huge pages in the page cache for other file systems.

We run the benchmarks inside virtual machines using the Kernel-based Virtual Machine (KVM) virtualization platform. Client threads, which generate traffic to the servers, are run outside the virtual machine, on the host OS. We run the client threads and server VM on the same system and use a bridge network with virtio between host and guest so that network performance is not a bottleneck. We isolate the CPU and memory of the guest VM and client threads on separate sockets using Linux's control group mechanism [38] to avoid performance interference. The benchmark VM is allocated 8 CPU cores, a typical medium-sized cloud instance. We set the Linux frequency governor to "performance" to disable dynamic CPU frequency changes during application runs.

4.2 Emulating slow memory: BadgerTrap

Dual-technology main memory, in particular Intel's 3D XPoint memory, is not yet available. Hence, we use a software technique to emulate slow memory while placing all data in conventional DRAM.

Each cache miss to slow memory should incur an access latency that is a multiple of the DRAM latency (e.g., 400ns slow memory [22] vs. 50ns DRAM latency). There is no easy mechanism to trap to software on all cache misses. Instead, we introduce extra latency by inducing page faults upon translation misses (TLB misses) to cold pages by using BadgerTrap [26].

The software fault mechanism is an approximation of an actual slow memory device. The BadgerTrap fault latency (about 1us in our kernel) is higher than some authors predict the 3D XPoint memory read latency will be [22]. Furthermore, the poisoned PTE will induce a fault even if the accessed memory location is present in the hardware caches. In these two respects, our approach over-estimates the penalty of slow memory accesses. However, once BadgerTrap installs a (temporary) translation, further accesses to other cache blocks on the same slow-memory page will not induce additional faults, potentially under-estimating impact. Our testing with micro benchmarks indicates our approach yields an average access latency to slow memory in the desired range, in part, because slow-page accesses are sufficiently infrequent that they nearly always result in both cache and TLB misses anyway, as we discuss in Section 3.3.

One important detail of our test setup is that we must install BadgerTrap (for the purpose of emulating slow mem-

ory latency) within the guest VM rather than the host OS. Thermostat communicates with the guest-OS BadgerTrap instance to emulate migration to slow memory. We must install BadgerTrap within the guest because, otherwise, each BadgerTrap fault would result in a `vmexit`. In addition to drastically higher fault latency, `vmexit` operations have the side-effect of changing the Virtual Processor ID (VPID) to 0. Since KVM uses VPIDs to tag TLB entries of its guests, installing a TLB entry with the correct VPID would entail complexity and incur even higher emulation latency. Since BadgerTrap on the guest entails a latency of $\approx 1\mu$s, which is already higher than projected slow-memory latencies [22], we did not want to incur additional slowdown by emulating slow memory in the host OS.

4.3 Benchmarks

We evaluate Thermostat with applications from Google's Perfkit Benchmarker and the Cloudsuite benchmarks [4, 25]. These applications are representative server workloads that have large memory footprints and are commonly run in virtualized cloud environments.

TPCC on MySQL: TPCC is a widely-used database benchmark, which aims to measure the transaction processing throughput of a relational database [7]. We execute TPCC on top of MySQL, one of the most popular open-source database engines, which is often deployed in the cloud. We use the open-source TPCC implementation from OLTP-Bench [20] (available at `https://github.com/oltpbenchmark/oltpbench`). We use a scale factor of 320, and run the benchmark for 600 seconds after warming up for 600 seconds. MySQL makes frequent I/O requests and hence benefits markedly from our use of `hugetmpfs` to enable huge pages for the OS page cache.

NoSQL databases: Aerospike, Cassandra, and Redis are popular NoSQL databases [1, 2, 5]. Cassandra is a wide-column database designed to offer a variable number of fields (or columns) per key, while Redis and Aerospike are simpler key-value databases that have higher peak throughput. Redis is single-threaded whereas Aerospike is multi-threaded. Cassandra performs frequent file I/O as it periodically compacts its SSTable data structure on disk [6]. So, Cassandra also benefits substantially from `hugetmpfs`. Redis performs no file I/O after loading its dataset into memory.

We tune Aerospike, Cassandra, and Redis based on the settings provided by Google's Perfkit Benchkmarker for measuring cloud offerings [4]. We use the YCSB traffic generator to drive the NoSQL databases [17]. For Aerospike we use 200M operations and for Cassandra we use 50M operations on 5M keys with 20 fields each with a Zipfian distribution. For both of these application, we evaluate two workload mixes: a read-heavy load with 95:5 read/write ratio and a write-heavy load with 5:95 read/write ratio. For Redis, we access keys with a hotspot distribution, wherein 0.01% of the keys account for 90% of the traffic. We vary value sizes according to the distribution reported in [12]. We observe 176K and 215K operations/sec for read-heavy and write-heavy workloads for Aerospike, and 21K and 45K operations/sec for read-heavy and write-heavy workloads for Cassandra. For Redis we observe 188K operations/sec for our baseline system with all pages in DRAM as huge pages.

	Resident Set Size	File-mapped
Aerospike	12.3GB	5MB
Cassandra	8GB	4GB
MySQL-TPCC	6GB	3.5GB
Redis	17.2GB	1MB
In-memory-analytics	6.2GB	1MB
Web-search	2.28GB	86MB

Table 2. Application memory footprints: resident set size and file-mapped pages.

In-memory analytics: We evaluate Thermostat on in-memory analytics benchmarks from Cloudsuite [25]. In-memory analytics runs a collaborative filtering algorithm on a dataset of user-movie ratings. It uses the Apache Spark framework to perform data analytics. We set both executor and driver memory to be 6GB to execute the benchmark entirely in memory. We run the benchmark to completion, which takes 317 seconds for our baseline system with all pages in DRAM as huge pages.

Web search: Cloudsuite's web search uses the Apache Solr search engine framework. We run client threads on host and index nodes within the virtual machine. We set steady state time to be 300 seconds and keep default values for all the other parameters on the client machine. As specified by the benchmark, target response time requires 99% of the requests to be serviced in 200ms. For our baseline system with all pages in DRAM as huge pages, we observe 50 operations/sec with $\approx$ 85ms 99th percentile latency.

4.4 Runtime overhead of Thermostat Sampling

We measure the runtime overhead of Thermostat to ensure that application throughput is not degraded by Thermostat's page sampling mechanism. For sampling periods of 10s or higher, we observe negligible CPU activity from Thermostat and no measurable application slowdown ($< 1\%$).

5. Evaluation

We next measure Thermostat's automatic hot/cold classification and run-time placement/migration mechanisms on our suite of cloud applications.

Thermostat takes as input a tolerable slowdown; a single input parameter specified by a system administrator. It then automatically selects cold pages to be migrated to slow memory at runtime. We set a tolerable slowdown of 3% throughout our evaluation, since a higher slowdown may lead to an overall cost increase due to higher required CPU provisioning (which is more expensive than memory). Ther-

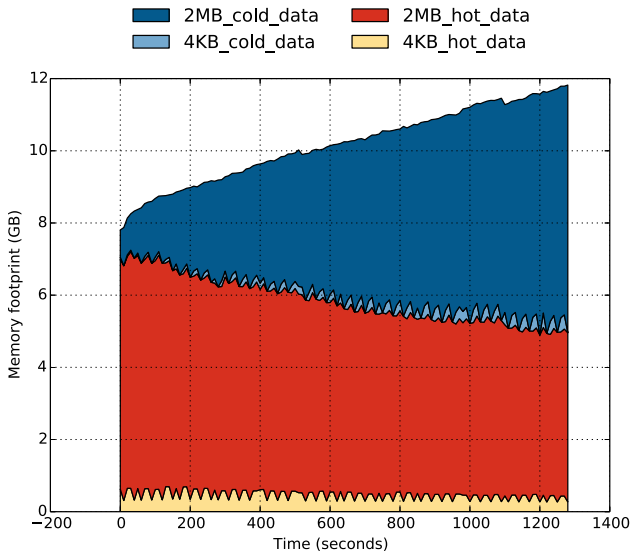

Figure 5. Amount of cold data in Cassandra identified at run time with 2% throughput degradation for a write-heavy workload (5:95 read/write ratio).

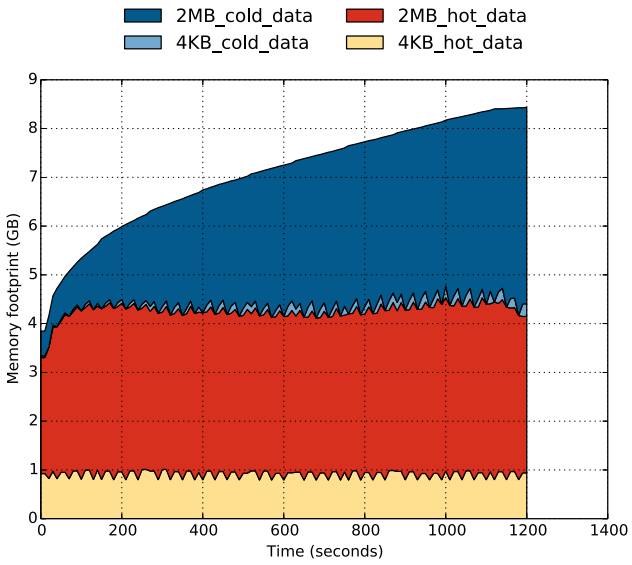

Figure 6. Amount of cold data in MySQL-TPCC identified at run time with 1.3% throughput degradation.

mostat's slowdown threshold can be changed at runtime through the Linux cgroup mechanism. Hence, application administrators can dynamically tune the threshold based on service level agreements for latency-critical applications or for the throughput requirements of batch jobs. We show that, for our application suite, Thermostat meets the target 3% slowdown while placing a significant fraction of application footprint in slow memory dynamically at runtime.

We evaluate Thermostat with 5% of huge pages sampled in every scan interval of 30s and at most 50 4KB pages poisoned for a sampled huge page. We compare the performance of Thermostat with a placement policy that places all pages in DRAM, which maximizes performance while incurring maximal memory cost. Thermostat's sampling mechanisms incur a negligible performance impact (well under 1%)—the slowdowns we report are entirely attributable to slow memory accesses.

We briefly discuss our findings for each application. Table 2 reports each application's footprint in terms of resident set size (RSS) and file-mapped pages. The memory savings quoted for each benchmark is the average cold memory fraction over the benchmark's runtime.

Cassandra: We report the breakdown of hot and cold 2MB and 4KB pages over time for Cassandra in Figure 5 for the write-heavy workload. Thermostat identifies between 40-50% of Cassandra's footprint (including both 4KB and 2MB pages) as cold. Note that the cold 4KB pages are solely due to splitting of cold huge pages during profiling ($\approx$ 5% of cold pages are 4KB, since our profiling strategy is agnostic of a page being hot or cold). Note that the resulting throughput degradation of 2% falls slightly under our target of 3%. We observe $\approx$ 1% higher average, 95th, and 99th

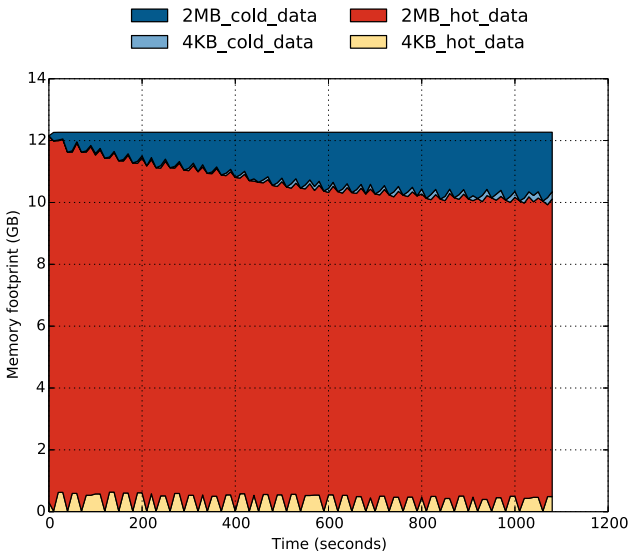

Figure 7. Amount of cold data in Aerospike identified at run time with 1% throughput degradation for a read-heavy workload (95:5 read/write ratio).

percentile read/write latency for Cassandra with Thermostat. Based on this performance vs. footprint trade-off, we estimate Thermostat enables a net cost savings of $\approx$ 30% for Cassandra (see Section 5.3 for a detailed analysis). For the read-heavy workload Thermostat identifies 40% of data as cold with 2.5% throughput degradation (we omit figure due to space constraints).

The memory consumption of Cassandra grows due to in-memory Memtables filling up. The Memtable is flushed to disk in the form of an SSTable, which then leads to a

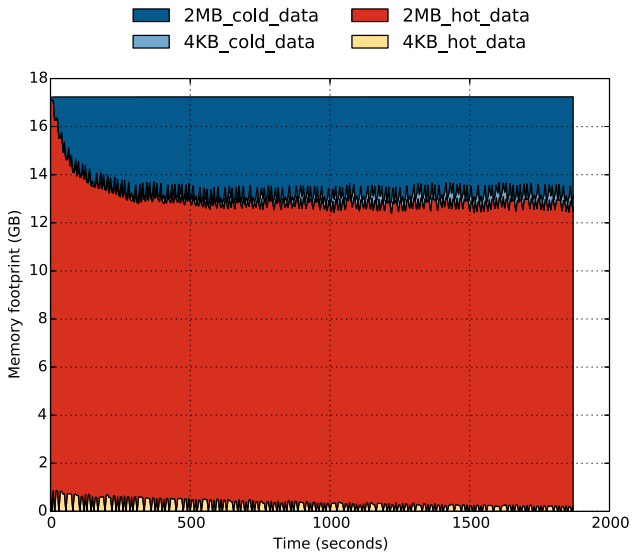

Figure 8. Amount of cold data in Redis identified at run time with 2% throughput degradation.

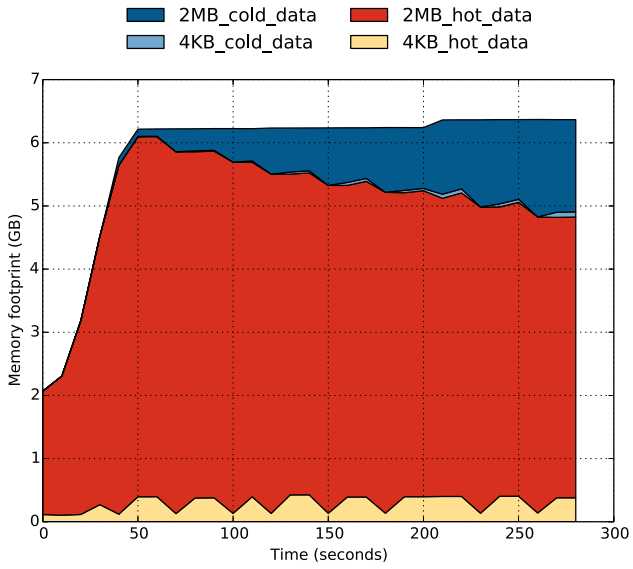

Figure 9. Amount of cold data in in-memory analytics benchmark identified at run time with 3% runtime overhead.

sharp decrease in memory consumption. However, we do not observe such a compaction event in our configuration, due to the large amount of memory provisioned for Cassandra in our test scenario.

MySQL-TPCC: In Figure 6 we show a similar footprint graph for MySQL-TPCC. The largest table in the TPCC schema, the LINEITEM table, is infrequently read. As a result, much of TPCCs footprint (about 40-50%) is cold and can be placed in slow memory while limiting performance degradation to 1.3%.

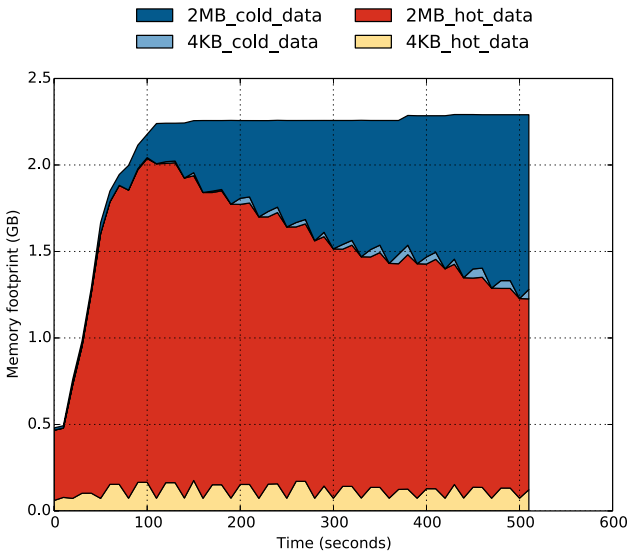

Figure 10. Amount of cold data in web-search benchmark identified at run time with 1% throughput and no 99th percentile latency degradation.

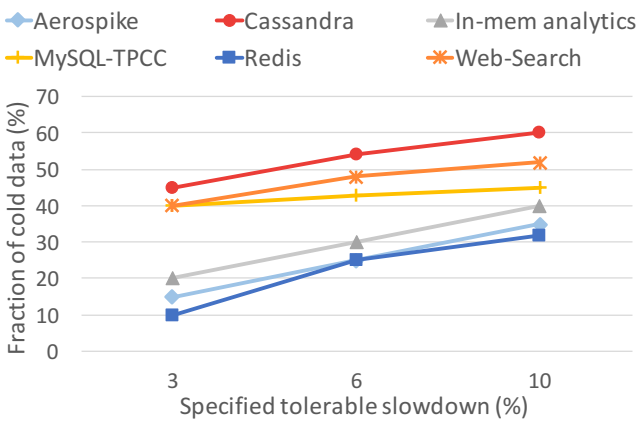

Figure 11. Amount of cold data in identified at run time varying with the specified tolerable slowdown. All the benchmarks meet their performance targets (not shown in the figure) while placing cold data in the slow memory.

Aerospike: In Figure 7 we show a similar footprint graph for Aerospike for the read-heavy workload. We see a small fraction of the footprint (about 15%) identified as cold while maintaining the tolerable slowdown. The average, 95th and 99th read/write latencies are all within 3% of the baseline. For the write-heavy workload, Thermostat identifies about 15% of data as cold while satisfying tolerable slowdown (we omit figure due to space constraints).

Redis: Unlike the other applications, Redis has a more uniform access pattern. The key data structure in Redis is a large hash table, hence, memory accesses are spread relatively uniformly over its address space; the relative hotness of pages reflects the corresponding hotness of the key dis-

tribution. We study a load where 0.01% of keys account for 90% of accesses. In Figure 8 we show that, under this load, 10% of the data is detected as cold at a 3% throughput degradation. The average read/write latency is 3.5% higher than the baseline.

In-memory analytics: We also evaluate in-memory analytics benchmark from Cloudsuite [25]. In Figure 9 we show Thermostat detects about 15-20% data as cold. As application footprint grows, Thermostat scans more pages and thus the cold page fraction also grows with time. We run this benchmark to completion, however, the benchmark runtime is much shorter than the previous data serving applications. (Cloudsuite is designed for tractable runtimes under expensive instrumentation and/or simulation). Nevertheless, Thermostat successfully identifies cold data while meeting the slowdown target. We expect the cold memory footprint of this application to reach steady state if a larger input were available.

Web search: In Figure 10 we show the footprint graph for the web search workload. We see about 40% of the footprint identified as cold. We observe $< 1\%$ degradation in throughput and no observable degradation in 99th percentile latency of 200ms.

5.1 Sensitivity to tolerable slowdown target

Next we show the sensitivity of Thermostat to the single input parameter specified by a system administrator, the tolerable slowdown. In our baseline evaluation, we set this parameter to 3%. However, due to changes in the price point of the memory technology or changes in data-center cost structure it may be possible to tolerate higher slowdown. To study the adaptability of Thermostat in such scenarios we also evaluate 6% and 10% slowdown targets. We show the variation in amount of cold data identified by Thermostat at run time with tolerable slowdown in Figure 11.

We observe that with increase in tolerable slowdown Thermostat can place a higher fraction of memory footprint in slow memory. We also observe that the performance targets of all applications are met (we omit data due to space constraints). However, in several cases, the achieved slowdown is less than the specified slowdown.

For MySQL-TPCC, Thermostat is not able to identify additional cold data even with an increase in tolerable slowdown (cold data fraction saturates at $approx$ 45%). This saturation happens because all remaining pages for TPCC are highly accessed, and placing any of them in slow memory results in an unacceptable application slowdown. For Aerospike, Thermostat is able to scale the cold data with varying tolerable slowdown. However, the actual slowdown doesn't reach the target slowdown due to (a) Aerospike's performance being insensitive to cold page accesses, and (b) the average OS fault handler latency for emulation being lower than the assumed latency of 1us used in Thermostat.

In summary, Thermostat places a higher fraction of data in slow memory if the user can tolerate more slowdown.

(MB/s)	Migration	False-classification
Aerospike	13.3	9.2
Cassandra	9.6	3.8
In-mem-Analytics	16	0.4
MySQL-TPCC	6	1.8
Redis	11.3	10
Web-search	1.6	0.3

Table 3. Data migration rate and false classification rate of slow memory. These rates are well-below expected bandwidth of near-future memory technologies.

Slow memory cost relative to DRAM	0.33×	0.25×	0.2×
Aerospike	10%	11%	12%
Cassandra	27%	30%	32%
In-mem-Analytics	11%	12%	13%
MySQL-TPCC	27%	30%	32%
Redis	17%	19%	20%
Web-search	27%	30%	32%

Table 4. Memory spending savings relative to an all-DRAM system when using slow memory with different cost points relative to DRAM.

This feature allows system administrators to better utilize expensive DRAM capacity by moving as much cold data to slow memory as possible via Thermostat.

5.2 Migration overhead and slow memory access rate

To verify that Thermostat doesn't cause un-realizable bandwidth pressure on the slow memory, we measured the memory bandwidth required by migrations and false classifications between slow and fast memory. In Table 3 we observe that the required migration bandwidth is < 30 MB/s on average across all benchmarks. The highest total traffic to cold memory we observe is 60 MB/s, which is well within the projected capability of near-future cheap memory technologies [22]. Thus, we infer that Thermostat doesn't produce unrealistic pressure on the memory system.

5.3 DRAM Cost Analysis

Since DRAM pricing is volatile, and slow memory prices remain unclear, it is difficult to perform a rigorous cost-savings analysis for Thermostat. We use a simple model to estimate the cost-savings possible with Thermostat and study a range of possible ratios of DRAM to slow-memory cost. Table 4 shows the fraction of DRAM spending saved due to Thermostat when slow memory is $\frac{1}{3}$, $\frac{1}{4}$ and $\frac{1}{5}$ of DRAM cost. We can see that, depending on workload and memory technology, anywhere from $\approx 10\%$ (for Aerospike) to 32% (for Cassandra) of DRAM cost can be saved.

6. Discussion

6.1 Hardware support for access counting

The software-only page access counting mechanism described in Section 3.3 is desirable since it can be run on *current* commodity x86 hardware. However it has two sources of inaccuracy: (i) we can only count TLB misses instead of LLC misses, and (ii) the measurement process itself throttles accesses to the poisoned pages, since the poison faults to the same page are serialized. We describe below two extensions to x86 hardware that can address these shortcomings.

6.1.1 Fault on LLC miss:

To accurately count the number of actual cache misses caused by a page, the x86 PTE can be modified to include a "count miss" (CM) bit. The CM bit should be stored in the corresponding TLB entry as well. When the CM bit is set in a translation entry, any LLC miss to that page will result in a software fault. The CM fault handler can then be used to track the number of cache misses to that page in the same way that BadgerTrap uses the reserved bit fault handler to track number of TLB misses. The instruction triggering a CM fault should retire once the data is fetched, since replaying the instruction could lead to a cascade of CM faults. During a CM fault, the actual memory (DRAM or NVRAM) access can be done in parallel with servicing the fault to partially or fully hide the CM fault service latency.

6.1.2 PEBS based access counting:

In this scheme, the PEBS (Precise Event Based Sampling) subsystem in the x86 architecture [30] can be extended to record page access information. In the current PEBS implementation, a PEBS record is stored in a pre-defined memory area on samples of specific events (LLC misses is one of them). When the area fills up, an interrupt is raised, which the kernel can then service and inspect the instructions that led to those events.

The maximum PEBS sampling frequency is limited by how fast the memory area can be filled and kernel interrupts serviced. The default value in the Linux kernel for PEBS sampling frequency is 1000Hz, which is far too low to support $\approx$ 30,000 slow memory accesses that can be done by a single thread for a 3% performance slowdown. If the record entry *only* stores the physical page address of the access, it can be stored in 48b, which is far less than the entire CPU state.

Merits to slow memory software-emulation: One of the major challenges in deploying new hardware in data centers is to evaluate impact on throughput degradation and tail latency to avoid violating service level agreements. Thermostat can be used in test nodes of production systems today to evaluate the performance implication of deploying slow memory in data centers. Thermostat does not need any specialized test hardware and is pluggable with a parameterized delay for simulating slow memory. Approaches without extensive hardware changes are likely to receive more widespread adoption in the industry [10]. Thus, using our evaluation system, we can easily evaluate the impact of using slow memory in an application for a given traffic pattern. For example, we experimented with a Zipfian traffic pattern for Redis and failed to place more that 10% of its footprint in slow memory without significant throughput degradation. Thus, such a tool allows one to evaluate the potential usability of slow memory in production *without any extra software instrumentation or hardware investment*.

Device wear: Some likely candidates for cheap, slow memory technologies are subject to device wear, which means that the memory may not function properly if it is written too frequently. Qureshi et al. propose a simple wear-leveling technique that uses an algebraic mapping between logical addresses and physical addresses along with address-randomization techniques to improve the lifetime of memory devices subject to wear [40]. Table 3 shows that accesses to slow memory by Thermostat fall well below the expected endurance limits of future memory technologies.

Spreading a 2MB page across fast and slow memories: In our scheme, the entirety of a 2MB is placed in slow or fast memory. This has the benefit of reducing TLB misses, but consumes extra fast memory space for 2MB pages with a small hot footprint. The evaluation of a scheme which selectively places only hot portions of an otherwise cold 2MB page in fast memory is left for future work.

7. Related Work

Application-guided two-level memory: Dulloor et al. [22] propose X-Mem, a profiling based technique to identify data structures in applications that can be placed in NVM. To use X-Mem, the application has to be modified to use special `xmalloc` and `xfree` calls instead of `malloc` and `free` and annotate the data structures using these calls. Subsequently, the X-Mem profiler collects a memory access trace of the application via PinTool [37], which is then post-processed to obtain access frequencies for each data structure. Using this information, the X-Mem system places each data structure in either fast or slow memory.

Such an approach works well when: (1) an overwhelming majority of the application memory consumption is allocated by the application itself, (2) modifiable application source code is available along with a deep knowledge of the application data structures, and (3) a representative profile run of an application can be performed. Most cloud-computing software violates some, or all, of these criteria. As we show in Section 5, NoSQL databases interact heavily with the OS page cache, which accounts for a significant fraction of their memory consumption. Source code for cloud applications is not always available. Moreover, even when source code is available, there is often significant variation in hotness within data structures, e.g., due to hot keys in a key-value store. Obtaining representative profile runs is difficult due to

high variability in application usage patterns [12]. In contrast, Thermostat is application transparent, can dynamically identify hot and cold regions in applications at a page granularity (as opposed to data structure granularity), and can be deployed seamlessly in multi-tenant host systems.

Lin et al. present a user-level API for memory migration and an OS service to perform asynchronous, hardware-accelerated memory moves [36]. However, Thermostat is able to avoid excessive page migrations by accurately profiling and identifying hot and cold pages, and hence can rely on Linux's existing page migration mechanism without suffering undue throughput degradation. Agarwal et al. [8, 9] demonstrate the throughput advantages of profile guided data placement for GPU applications. Chen et al. [16] present a portable data placement engine directed towards GPUs. Both of these proposals seek to maximize bandwidth utilization across memory sub-systems with disparate peak bandwidth capability. They focus on bandwidth- rather than latency-sensitive applications and do not seek to reduce system cost.

Linux provides the `madvise` API for applications to provide hints about application memory usage. Jang et al. propose an abstraction for *page coloring* to enable communication between applications and the OS to indicate which physical page should be used to back a particular virtual page. Our approach, however, is application transparent and does not rely on hints, eliminating the programmer burden of rewriting applications to exploit dual-technology memory systems.

Software-managed two-level memory: AutoNUMA [18] is an automatic placement/migration mechanism for co-locating processes and their memory on the same NUMA node to optimize memory access latency. AutoNUMA relies on CPU-page access affinity to make placement decisions. In contrast, Thermostat makes its decisions based on the page access rate, irrespective of which CPU issued the accesses.

Hardware-managed two-level memory: Several researchers [23, 39, 41] have proposed hybrid dual technology memory organizations with hardware enhancements. In such approaches, DRAM is typically used as a transparent cache for a slower memory technology. Such designs require extensive changes to the hardware memory hierarchy; Thermostat can place comparable fractions of the application footprint in slower memory without any special hardware support in the processor or caches.

Disaggregated memory: Lim et al. [34, 35] propose a disaggregated memory architecture in which a central pool of memory is accessed by several nodes over a fast network. This approach reduces DRAM provisioning requirements significantly. However, due to the high latency of network links, performing remote accesses at cache-line level granularity is not fruitful, leading Lim to advocate a remote paging interface. For the latency range that Lim assumed (12-15us), our experience confirms that application slowdowns are prohibitive. However, for the expected sub-microsecond access latency of near-future cheap memory technologies, our work shows that direct cache-line-grain access to slow memory can lead to substantial cost savings.

Cold data detection: Cold/stale data detection has been extensively studied in the past, mainly in the context of paging policies and disk page caching policies, including mechanisms like `kstaled` [32, 42, 44]. Our work differs from such efforts in that we study the impact of huge pages on cold data detection, and show a low-overhead method to detect and place cold data in slow memory without significant throughput degradation. Baskakov et al. [14] and Guo et al. [28] propose a cold data detection scheme by breaking 2MB pages into 4KB pages so as to detect cold-spots in large 2MB pages utilizing PTE "Accessed" bits. However, as we have shown in Section 2.1, obtaining performance degradation estimates from Accessed bits is difficult. Thus, we instead use finer grain access frequency information obtained through page poisoning.

NVM/Disks: Several upcoming memory technologies are non-volatile, as well as slower and cheaper than DRAM. The non-volatility of such devices has been exploited by works like PMFS [21] and pVM [31]. These works shift a significant fraction of persistent data from disks to non-volatile memories (NVMs), and obtain significant performance benefits due to the much faster speed of NVMs as compared to disks. Our work explores placing volatile data in cheaper memory to reduce cost, and as such is complementary to such approaches.

Hardware modifications for performance modeling: Li et al. [33] propose a set of hardware modifications for gauging the impact of moving a given page to DRAM from NVM based on its access frequency, row buffer locality and memory level parallelism. The most impactful pages are then moved to DRAM. In a similar vein, Azimi et al. [13] propose hardware structures to gather LRU stack and miss rate curve information online. However, unlike our work, these techniques require several modifications to the processor and memory controller architecture. Existing performance counter and PEBS support has been utilized by prior works [24, 43] to gather information related to the memory subsystem. However, gathering page granularity access information at higher frequency from the PEBS subsystem requires a high overhead and so is not desirable.

8. Conclusion

With recent announcements of 3D XPoint memory by Intel/Micron there is an opportunity of cutting cost of memories in data centers with improved capacity and cost per bit. However, due to projected higher access latencies of these new memory technologies it is not feasible to completely replace main memory DRAM technology. To address the renewed interest in two-tiered physical memory we presented and evaluated Thermostat, an application-transparent huge-

page-aware mechanism to place pages in a dual technology hybrid memory system, while achieving both the cost advantages of two-tiered memory and performance advantages of transparent huge pages. Huge pages, being performance critical in cloud applications with large memory footprints, especially in virtualized cloud environments, need to be supported in this two-tier memory system. We present a new hot/cold classification mechanism to distinguish frequently accessed pages (hot) from infrequently accessed ones (cold). We implement Thermostat in Linux kernel version 4.5 and show that it can transparently move cold data to slow memory while satisfying a 3% tolerable slowdown. We show that our online cold page identification mechanism incurs no observable performance overhead and can migrate up to 50% of application footprint to slow memory while limiting performance degradation to 3%, thereby reducing memory cost up to 30%.

9. Acknowledgements

We would like to thank Google for providing financial support for this work. We would also like to thank Subhasis Das and Akshitha Sriraman for their valuable feedback on drafts of this manuscript.

References

[1] Aerospike. `http://www.aerospike.com/`. [Online; accessed 8-Aug-2016].

[2] Cassandra. `http://cassandra.apache.org/`. [Online; accessed 8-May-2016].

[3] Libvirt virtualization api. `http://libvirt.org/formatdomain.html#elementsCPU`. [Online; accessed 13-Aug-2016].

[4] PerfKit Benchmarker. `https://github.com/GoogleCloudPlatform/PerfKitBenchmarker`. [Online; accessed 8-May-2016].

[5] Redis. `http://redis.io/`. [Online; accessed 8-May-2016].

[6] MemtableSSTable. `https://wiki.apache.org/cassandra/MemtableSSTable`. [Online; accessed 8-May-2016].

[7] TPC-C. `http://www.tpc.org/tpcc/`. [Online; accessed 8-May-2016].

[8] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch. Unlocking Bandwidth for GPUs in CC-NUMA Systems. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 354–365, February 2015.

[9] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 607–618, March 2015.

[10] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler. Selective GPU caches to eliminate CPU-GPU HW cache coherence. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 494–506, Mar. 2016.

[11] AMD. AMD-V Nested Paging . `http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf`, 2008. [Online; accessed 2-May-2016].

[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, June 2012.

[13] R. Azimi, L. Soares, M. Stumm, T. Walsh, and A. D. Brown. PATH: page access tracking to improve memory management. In *Proceedings of the International Symposium on Memory Management*, pages 31–42, Oct. 2007.

[14] Y. Baskakov, P. Gao, and J. K. Spencer. Identification of Low-activity Large Memory Pages. May 2016. URL `http://www.google.ch/patents/US9330015`. US Patent 9,330,015.

[15] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient Virtual Memory for Big Memory Servers. In *International Symposium on Computer Architecture (ISCA)*, pages 237–248, June 2013.

[16] G. Chen, B. Wu, D. Li, and X. Shen. PORPLE: An Extensible Optimizer for Portable Data Placement on GPU. In *International Symposium on Microarchitecture (MICRO)*, pages 88–100, Dec. 2014.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of Symposium on Cloud Computing*, pages 143–154, June 2010.

[18] J. Corbet. AutoNUMA: the other approach to NUMA scheduling. `http://lwn.net/Articles/488709/`, 2012. [Online; accessed 9-May-2016].

[19] Dickins, Hugh. huge tmpfs: THPagecache implemented by teams. `https://lwn.net/Articles/682623/`, 2016. [Online; accessed 30-April-2016].

[20] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4): 277–288, 2013.

[21] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the European Conference on Computer Systems(EuroSys)*, pages 15:1–15:15, Apr. 2014.

[22] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 15:1–15:16, April 2016.

[23] M. Ekman and P. Stenstrom. A Cost-Effective Main Memory Organization for Future Servers. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 45a–45a, April 2005.

[24] S. Eranian. What Can Performance Counters Do for Memory Subsystem Analysis? In *International Conference on Archi-*

tectural Support for Programming Languages and Operating Systems (ASPLOS), pages 26–30, Mar. 2008.

[25] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, Mar. 2012.

[26] J. Gandhi, A. Basu, M. H. Hill, and M. M. Swift. A Tool to Instrument x86-64 TLB Misses. *SIGARCH Computer Architecture News (CAN)*, 2014.

[27] M. Gorman. Huge pages part 4: benchmarking with huge pages. `http://lwn.net/Articles/378641/`, 2010. [Online; accessed 11-Aug-2016].

[28] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. In *International Conference on Virtual Execution Environments(VEE)*, pages 39–51, Mar. 2015.

[29] Intel. Intel and Micron Produce Breakthrough Memory Technology. `https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/`, 2015. [Online; accessed 29-April-2016].

[30] Intel. Intel 64 and IA-32 Architectures Developer's Manual, 2016. [Online; accessed 2-May-2016].

[31] S. Kannan, A. Gavrilovska, and K. Schwan. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *Proceedings of the European Conference on Computer Systems(EuroSys)*, pages 13:1–13:16, Apr. 2016.

[32] Lespinasse, Michel. idle page tracking / working set estimation. `https://lwn.net/Articles/460762/`, 2011. [Online; accessed 29-April-2016].

[33] Y. Li, J. Choi, J. Sun, S. Ghose, H. Wang, J. Meza, J. Ren, and O. Mutlu. Managing Hybrid Main Memories with a Page-Utility Driven Performance Model. *arXiv preprint arXiv:1507.03303*, 2015.

[34] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *International Symposium on Computer Architecture (ISCA)*, pages 267–278, June 2009.

[35] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level Implications of Disaggregated Memory. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, Feb. 2012.

[36] F. X. Lin and X. Liu. Memif: Towards Programming Heterogeneous Memory Asynchronously. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 369–383, Apr. 2016.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation(PLDI)*, pages 190–200, 2005.

[38] Menage, Paul. CGROUPS . `http://lxr.free-electrons.com/source/Documentation/cgroup-v1/memory.txt`. [Online; accessed 4-May-2016].

[39] A. Mirhosseini, A. Agrawal, and J. Torrellas. Survive: Pointer-based In-DRAM Incremental Checkpointing for Low-Cost Data Persistence and Rollback-Recovery. *IEEE Computer Architecture Letters*, PP(99):1–1, 2016.

[40] M. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling. In *International Symposium on Microarchitecture (MICRO)*, Nov. 2009.

[41] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *International Symposium on Computer Architecture (ISCA)*, pages 24–33, June 2009.

[42] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Operating Systems Review*, 36(SI):181–194, Dec. 2002.

[43] T. Walsh. Generating Miss Rate Curves with Low Overhead Using Existing Hardware. Master's thesis, University of Toronto, 2009.

[44] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 177–188, Oct. 2004.