"Anti-Caching"-based Elastic Memory Management for Big Data

Hao Zhang #, Gang Chen †‡, Beng Chin Ooi #, Weng-Fai Wong # Shensen Wu †‡, Yubin Xia *

School of Computing, National University of Singapore, Singapore, {zhangh, ooibc, wongwf}@comp.nus.edu.sg

† *College of Computer Science, Zhejiang University, China,* ‡ *yzBigData Co., Ltd., China,* †‡ *{cg, shensenwu}@cs.zju.edu.cn*

* *Institute of Parallel and Distributed Systems (IPADS), Shanghai Jiao Tong University, China, xiayubin@sjtu.edu.cn*

***Abstract*—The increase in the capacity of main memory coupled with the decrease in cost has fueled the development of in-memory database systems that manage data entirely in memory, thereby eliminating the disk I/O bottleneck. However, as we shall explain, in the Big Data era, maintaining all data in memory is impossible, and even unnecessary. Ideally we would like to have the high access speed of memory, with the large capacity and low price of disk. This hinges on the ability to effectively utilize both the main memory and disk.**

In this paper, we analyze state-of-the-art approaches to achieving this goal for in-memory databases, which is called as "Anti-Caching" to distinguish it from traditional caching mechanisms. We conduct extensive experiments to study the effect of each fine-grained component of the entire process of "Anti-Caching" on both performance and prediction accuracy. To avoid the interference from other unrelated components of specific systems, we implement these approaches on a uniform platform to ensure a fair comparison. We also study the usability of each approach, and how intrusive it is to the systems that intend to incorporate it. Based on our findings, we propose some guidelines on designing a good "Anti-Caching" approach, and sketch a general and efficient approach, which can be utilized in most in-memory database systems without much code modification.

I. Introduction

Data is invaluable in product prediction, scientific exploration, business intelligence, and so on. However, the sheer quantity and velocity of Big Data have caused problems in data capturing, storage, maintenance, analysis, search, and visualization. The management of a huge amount of data is particularly challenging to the design of database architectures. In addition, in many instances when dealing with Big Data, speed is not an option but a must. For example, Facebook makes an average of 130 internal requests sequentially for generating the HTML for a page [1], thus making long-latency data access unacceptable. Supporting ultra-low latency service is therefore a requirement. Effective and smart decision making is enabled with the utilization of Big Data, however, on the condition that real-time analytics is possible. Otherwise, profitable decisions could become stale and useless. Therefore, efficient real-time data analytics is important and necessary for modern database systems.

Distributed and NoSQL databases have been designed for large scale data processing and high scalability [2], [3], while the Map-Reduce framework provides a parallel and robust solution to complex data computation [4], [5]. Synchronous Pregel-like message-passing [6] or asynchronous GraphLab [7] processing models have been utilized to tackle large graph analysis, and stream processing systems [8], [9] have been designed to deal with the high velocity of data generation. Recently, in-memory database systems have gained traction as a means to significantly boost the performance.

A. Towards In-memory Databases

The performance of disk-based databases, slowed down by unpredictable and high access latency of disks, is no longer acceptable in meeting the rigorous low-latency, real-time demands of Big Data. The performance issue is further exacerbated by the overhead (e.g., system calls, buffer manager) hidden by the I/O flow. To meet the strict real-time requirements for analyzing massive amount of data and servicing requests within milliseconds, an in-memory database that keeps data in the main memory all the time is a promising alternative.

Jim Gray's insight that "memory is the new disk, disk is the new tape" is becoming true today – we are witnessing a trend where memory will eventually replace disk and the role of disk must inevitably become more archival in nature. Memory capacity and bandwidth have been doubled every two years, while its price has been dropping by a factor of 10 every five years. In the last decade, the availability of large amounts of DRAM at plummeting cost helped to create new breakthroughs, making it viable to build in-memory databases where a significant part, if not the entirety, of the database fits in the main memory.

In-memory databases have been studied as early as the 80s [10], [11], [12], [13]. Recent advances in hardware technology re-kindled the interest in implementing in-memory databases as a means to provide faster data accesses and real-time analytics [14], [15], [16], [17]. Most commercial database vendors (e.g., SAP, Microsoft, Oracle) have begun to introduce in-memory databases to support large-scale applications completely in memory [18], [19], [20], [21]. Nevertheless, in-memory data management is still at its infancy with many challenges [22], [23], [24], [25], and a completely in-memory design is not only still prohibitively expensive, but also unnecessary. Instead, it is important to have a mechanism for in-memory databases that utilize both memory and disks effectively. It is similar to the traditional caching process, which is however the other way around: instead of fetching data that is needed from disk into main memory, cold data is evicted to disk, and fetched again only when needed. In this case, main memory is treated as the main storage, while

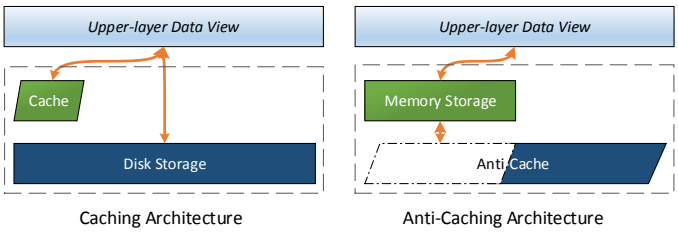


Fig. 1: Caching vs "Anti-Caching"

disk acts as a backup. We call it as "Anti-Caching"[1], to emphasize the opposite direction of data movement. The goal of "Anti-Caching" is to enable in-memory databases to have the "capacity as data, speed as memory and price as disk" [27], [28], being a hybrid and alternative between strictly memory-based and disk-based databases.

B. Contributions and Paper Organization

The major contributions of this paper are:

1) We investigate various "Anti-Caching" approaches that are designed for use in both user-space and kernel-space, and their tradeoffs in terms of both performance and usability.
2) An experimental study is conducted to analyze the components of the "Anti-Caching" process, in terms of the overhead involved, eviction accuracy, I/O efficiency, CPU utilization and other customized optimizations exclusively for certain scenarios.
3) We present some implications for designing an "Anti-Caching" approach, and also sketch a general approach trying to combine the advantages of being in user-space and kernel-space.

The remainder of this paper is structured as follows. We give a discussion about the differences between caching and "Anti-Caching" in Section II. In Section III, we present state-of-the-art "Anti-Caching" approaches, and their basic mechanisms. We conduct a thorough experimental analysis of these approaches, with regards to the performance effect on the systems in Section IV. Section V summarizes what we derive from our experimental analysis, and presents some insights and tradeoffs on designing an "Anti-Caching" approach. Based on our findings, we sketch an easy-to-use general approach that can be incorporated into most in-memory databases easily in Section VI. Section VII concludes the paper.

II. Caching vs "Anti-Caching"

In general, both caching and "Anti-Caching" are utilized to deal with the gap between main memory and disk in terms of speed and capacity. Caching is to cache the disk data in memory to speed up the data access performance, while "Anti-Caching" is to "anti-cache" the in-memory data onto disk to extend its storage capacity, as depicted in Figure 1. At the first glance, they share common goals and deal with the same level of storages (i.e., main memory and disk), so that identical mechanisms can be applied to both, which, however, is not true in practice. We summarize the key differences between caching and "Anti-Caching" as follows.

- The fundamental assumptions about the memory size are different. In the caching domain, the memory size is much smaller than the total data size, while in the "Anti-Caching" domain, the memory size is relatively large, which means it can hold a significant portion of the data. The distinctive assumptions make a difference in the system design, as with larger available memory, we can have more flexibility to manage the memory (e.g., we can sacrifice more memory overhead for better performance; we can delay data eviction because there is no tight memory bound).
- They are targeted for different types of systems. That is, caching is for disk-based systems while "Anti-Caching" is for in-memory systems, which is more overhead-sensitive. This makes the presumed negligible overhead caused by caching (e.g., access tracking) in disk-based systems a visible performance influence factor for in-memory systems. Moreover, disk I/O cannot become the system bottleneck again for in-memory systems.
- The primary data location is different, i.e., disk for caching but main memory for "Anti-Caching", so that in "Anti-Caching", disk is treated as an extended temporary storage for evicted data, not the primary host for the whole data. Ideally, there is no need for data format transformation in "Anti-Caching" because it is better to keep the data in memory-optimized format even on disk to fasten the loading and transforming process [29], in contrast to the disk-optimized format as traditional disk-based systems.

III. State-of-the-art Approaches

In general, state-of-the-art "Anti-Caching" approaches can be classified into three categories.

1) User-space: "Anti-Caching" performed in the user-space is the most commonly used approach. This enables ad hoc optimizations based on application semantics and fine-grained operations. On the other hand, the need to cross the OS boundary for I/O introduces additional overhead and constraints.

2) Kernel-space: Virtual memory management (VMM), which is available in most operating systems, can be regarded as a simple and general solution for "Anti-Caching" since the current generation of 64-bit OS (e.g., Linux, Windows 8.1) supports up to 128 TB (2^{47} bytes) of virtual address space, which is sufficient for most database applications. However, due to the lack of upper-layer application semantics, kernel-space solutions often suffer from inaccurate eviction decisions. Furthermore, the constraint of operating in units of pages when swapping data incurs extra overhead.

3) Hybrid of user- and kernel-space: A hybrid approach that combines the advantages from both semantics-aware user-space and hardware-assisted kernel-space approaches is promising. This can be done in either a user-centric or kernel-centric way. Such an approach would exploit the application's

[1]The term anti-caching was first used in [26], referring to the specific technique used in H-Store. Here we extend the term to mean a general mechanism that deals with data overflow for in-memory databases.

semantics as well as the efficiency provided by the OS in handling disk I/O, access tracking and book-keeping. It can also act as a general approach for most systems, rather than an ad hoc solution for a specific system.

In the following subsections, we will introduce some representative approaches in each category, and their basic mechanisms for "Anti-Caching". We break down "Anti-Caching" process into four major components, i.e., access pattern tracking, eviction strategy, book-keeping and data swapping (see Table I). In particular, access pattern tracking refers to the tracking of data access behavior in order to guide eviction decisions (e.g., LRU chain). An eviction strategy decides which data should be evicted under what conditions, based on the historical access information (e.g., LRU, FIFO, user-defined hot/cold classification). By book-keeping we mean the method(s) used to keep track of the location of the data (e.g., page table), especially that evicted to disk, while we call the process of writing evicted data to and read from disk as "data swapping" (e.g., page-level swapping, tuple-level direct reading/writing).

A. User-space Approaches

1) H-Store Anti-caching: H-Store [14], [23] is a distributed row-based in-memory relational database targeted for high-performance OLTP processing. Most of the heavy-weight components like locking, buffer management, which are commonly used in traditional disk-based databases, are removed. Anti-caching technique [26] has been proposed to overcome the restriction that all the data must fit in main memory. Basically, cold data is able to be moved to disk based on the LRU policy.

With full access to the operations within H-Store, the anti-caching component is able to track accesses in a fine-grained manner using its tuple-level LRU. In particular, it maintains an in-memory list of all tuples for each table in an LRU order, allowing for access to the least-recently-used tuples in real time. Both doubly-linked and singly-linked lists were experimented in H-Store, with a tradeoff between memory overhead and performance cost of maintaining the LRU order. Nevertheless, the overhead of maintaining the LRU list in the user-space is significant, but unavoidable, in terms of both memory consumption and performance degradation. With the fine-grained tracking mechanism, anti-caching is able to precisely evict the least-recently-used tuples.

To reduce the overhead of evicting, tuples are aggregated into groups of blocks which are then written out to disk in a single sequential write. Data fetching is also conducted in units of blocks, which, unfortunately, may waste a significant amount of I/O as redundant tuples are also fetched.

To track the status of a tuple (i.e., in-memory or on disk), an in-memory evicted table is used, and the index is updated accordingly based on the evicted table. Furthermore, given its targeted OLTP workload, some ad hoc optimizations can be applied. An example of such optimizations is non-blocking transactions, in which it simply aborts any transaction that accesses evicted data, and then restarts it at a later point when the data has been retrieved.

2) Siberia in Hekaton: Project Siberia [30], [31], [32] also adopts an "Anti-Caching" approach for Hekaton [21], which is a memory-optimized OLTP engine fully integrated into Microsoft SQL server. Hekaton is designed for high concurrency, and utilizes lock-free data structures and an optimistic multi-version concurrency control technique. Furthermore, Hekaton tables and regular tables can be accessed at the same time, thereby providing more flexibility to users.

Instead of maintaining an LRU list like H-Store, Siberia performs offline classification of hot and cold data by logging tuple accesses first, and then analyzing them offline to predict the top K hot tuples with the highest estimated access frequency, using an efficient parallel classification algorithm based on exponential smoothing [31]. The record access logging method incurs less overhead than an LRU list in terms of both memory and CPU. However, this offline method cannot detect rapid access fluctuations, and is not able to constrain memory usage below a given bound.

In addition, to relieve the memory overhead caused by the evicted tuples, Siberia does not store information in memory about the evicted tuples (e.g., keys in the index, evicted table) other than the compact Bloom and range filters [32] that are used to filter the access to disk. In fact, Siberia uses a separate store with simple storage functionality (e.g., insert, delete, retrieve) for cold tuples, which makes it necessary to transactionally coordinate between hot and cold stores so as to guarantee consistency [30]. Data in Hekaton is evicted to or fetched from the cold store on a tuple basis. In particular, eviction only occurs during data migration (from hot store to cold store) after a new result of classification feeds into the engine, while fetching can happen during migration (from cold store to hot store), and also referencing a tuple in the cold store. Only insert, update and migration can add tuples into the hot store; read does not affect the hot store. This is based on the assumption that the cold store is highly infrequently accessed.

3) Spark: Spark [16] features in-memory Map-Reduce data analytics with lineage-based fault-tolerance [39]. It presents a data abstraction – Resilient Distributed Dataset (RDD) – which is a coarse-grained deterministic immutable data structure. The ability of maintaining data in memory in a fault-tolerance manner makes RDD suitable for many data analytics applications, especially iterative jobs, since frequently-used data can be kept in memory instead of being shuffled to disk at each iteration.

In addition, RDDs can also be configured to be partially in-memory by allowing evictions at the level of partitions based on approximate LRU. In particular, when a new RDD partition is to be inserted into the memory store that lacks space, one or more least-recently-inserted partitions will be evicted. The partitions chosen to be evicted should not be within the same RDD as the newly-inserted one, because it assumes that most operations will run tasks over an entire RDD. These evicted partitions are put into the disk store, and the in-memory hash table is updated accordingly to reflect their storage status. There is no access tracking in Spark since its LRU is based on the insertion time, thus a simple linked-list would suffice.

4) Caching/Buffering: In traditional disk-based systems, caching/buffering is an important technique to alleviate the high-latency problem caused by disk storage. By keeping some data in a memory pool, frequently-used data can be retrieved from memory rather than from disk. In general, there are two kinds of caching/buffering mechanisms, i.e., general cache systems and ad hoc buffer management, which we shall elaborate below.

TABLE I: Summary of "Anti-Caching" Approaches

Approaches	Access Tracking	Eviction Strategy	Book-keeping	Data Swapping
H-Store anti-caching [26]	tuple-level tracking	LRU	evicted table and index	block-level swapping
Hekaton Siberia [30], [31], [32]	tuple-level access logging	offline classification	Bloom and range filter	tuple-level migration
Spark [16]	N/A	LRU based on insertion time	hash table	block-level swapping
Cache systems [33], [34]	tuple-level tracking	LRU, approximate LRU, etc	N/A	N/A
Buffer management [35]	page-level tracking	LRU, MRU, CLOCK, etc	hash table	page-level swapping
OS Paging [36]	h/w-assisted page-level tracking	LRU, NRU, WSCLOCK, PRRA, etc	page table	page-level swapping
Efficient OS Paging [37]	tuple-level access logging	offline classification and OS Paging	OS-dependent	OS-dependent
Access observer in HyPer [38]	h/w-assisted page-level tracking	N/A	N/A	N/A

a) General cache systems: There are some general cache systems with the key-value data model that can be used as a middle layer between databases and application servers, for caching frequently-used data in memory to reduce data access latency [40], [41]. Two representative widely-used cache systems are Memcached [33] and Redis [34].

Like H-store, access tracking and eviction in Memcached are based on an LRU list [26], while Redis provides a set of eviction strategies, from RANDOM to aging-based LRU. The latter is an approximate LRU (ALRU) scheme that samples a small number of keys, and evicting the one with the oldest access time. So instead of maintaining a full LRU list, Redis maintains an access time field (22 bits) for each tuple. In addition, a user-defined TTL (time-to-live) can also be used to control which tuple to evict and when to evict it. Due to the cache functionality provided by Memcached and Redis, they simply drop the evicted data rather than write it back to the disk.

b) Ad hoc buffer management: Traditional disk-based databases typically integrate buffer management into its data accesses in order to reduce the costly disk I/O [35]. Data is managed by the buffer manager in units of pages (also called frames), and the same granularity also applies to data swapping. Access indirection is used to track data access behavior, where a logical page ID is translated to a frame ID (i.e., virtual address). A pinning and unpinning sequence is used to request and release a page, and data access is only detected by the initial page request (i.e., pinning) from higher-level components. Thus, during the pinning and unpinning interval, multiple access operations within that page, is treated as one database page reference [35]. Different paging algorithms, such as LRU, MRU, FIFO, CLOCK, can be used depending on the implementations and configurations. For book-keeping, an in-memory hash table is used to keep track of the status of a page, and facilitate the translation between logical page ID and physical frame ID. In addition, it is also possible to prefetch a set of pages because databases can often predict the order in which pages will be accessed.

B. Kernel-space Approaches

1) OS Paging: Paging is an important part of virtual memory management (VMM) in most contemporary general-purpose operating systems (e.g., UNIX, LINUX, WINDOWS). It allows a program to process more data than the physically available memory [36]. In theory, a process can assume that it can use the entire available virtual address space (e.g., 2^{64} bytes in 64-bit OS) [2] as if it was in memory. The OS will swap in/out pages accordingly in a way that is totally transparent to the upper-layer user programs. This provides a uniform interface for users to manage their data, without complex consideration of where data is physically. This is a general approach that can be used by any system immediately with little implementation work, since most systems are built on top of the OS.

Just as its name implies, OS Paging tracks data access in the granularity of pages. It relies heavily on hardware (i.e., the memory management unit (MMU)) to record the access rather than purely software solutions. Specifically, all memory references must pass through the MMU, and get translated from virtual address to physical address. In the process, permissions are checked and appropriate flags (e.g., accessed, dirty) are set to indicate the access status for each page. As this is the part of the MMU's routine operations, VMM can get this information almost for free. In addition, these flags will be reset periodically by the OS based on its paging algorithm, capturing the latest access status. Some OSes also maintain custom flags (e.g., `PG_referenced` flag in LINUX) to record even more access information.

Various general paging algorithms such as LRU, MRU, CLOCK, NRU, WSCLOCK, can be used in OS Paging, but typically only one is chosen for a specific OS. For example, LINUX uses its customized PRRA (Page Frame Reclaiming Algorithm), which maintains *active* and *inactive* doubly-linked lists so as to perform LRU operations. The *active* list tends to include the pages that have been accessed recently, while the *inactive* list tends to include those that have not been accessed for some time. The movements between *active* and *inactive* lists are conducted periodically or on demand, and pages can only be reclaimed from the *inactive* list [42].

To track the location of a page, the OS marks a swapped-out page as "not present in memory" and record its disk location in the Page Table (i.e., clear the `Present` flag in the Page Table entry, and encodes the location of the swapped-out page in the remaining bits). This keeps the additional memory overhead to a minimum. Any access to a swapped-out page triggers a page fault. This will be caught by the OS and resolved by swapping in the required page.

In general, there are two methods that we can use in order to utilize OS Paging. One natural way is to configure a swap partition/file, which will enable OS Paging automatically without any programming effort on the applications. The other

[2] In practice, the available virtual address space may be limited to 2^{47} bytes or less.

way is to use memory-mapped files. The latter maps a file on disk onto a region in virtual memory space. File reads/writes are then achieved by direct memory access as if all the data was in memory, during which the OS will transparently page in/out the necessary data. Memory-mapped file technique is widely used in the database area. Popular examples include MongoDB [43], MonetDB [44], and MDB [45].

C. Hybrid of User- and Kernel-space Approaches

In general, user-space approaches can take advantage of application semantics to enable ad hoc optimizations and fine-grained operations. In contrast, kernel-space approaches deal better with I/O scheduling and can utilize hardware-assisted mechanisms. We introduce two approaches that try to combine the advantages from both user- and kernel-spaces.

1) Efficient OS Paging: In [37], a more efficient OS paging based on separating hot and cold data was proposed. The hot memory region is `mlock`-ed while the cold region is left exposed to OS Paging. This also helps reduce the possibility that victim data would be evicted/fetched redundantly because it would be in the same page as the other data. Data accesses are tracked using access logging, and then processed offline. The offline classification of hot/cold data is based on the exponential smoothing algorithm [31]. The book-keeping and data swapping are then handled by the OS VMM without any involvement from user applications.

2) Access observer: In [38], an access observation method to assist their hot/cold data classification was proposed. It is used to compact the data in memory thereby reducing memory usage pressure, as well as to support memory-consumption-friendly snapshotting. Specifically, there are two approaches to observing the data access at the page-level. One approach is to use the *mprotect* system call to prevent accesses to a range of virtual memory pages. Whenever there is an access to that region, a SIGSEGV signal is caught by a registered signal handler that will record the access. Alternatively, a more efficient way is to use hardware-assisted access tracking as the OS VMM. User-space APIs can be provided to manage page table directly from user-space. We will elaborate these two approaches in Section IV.

D. Summary

In summary, existing approaches adopt different strategies for each component of "Anti-Caching", with different constraints and tradeoffs, in terms of memory and CPU overhead, usability, generality, etc. Semantic information available to user-space approaches in some systems enables ad hoc optimizations and finer-grained access tracking and book-keeping. On the other hand, kernel-space approaches have the advantage of being hardware-assisted (e.g., page access tracking, virtual address translation), having little overhead while being general enough to be used in most systems. We will further investigate different strategies for each component respectively in Section IV, in order to see what really happens under the hood, and obtain some implications on choosing different strategies in designing a good "Anti-Caching" approach.

IV. Under the Hood: An Experimental Analysis

In this section, we shall empirically study the different possible strategies for each component in the "Anti-Caching" process, namely, access tracking, eviction strategy, book-keeping and data swapping, in terms of memory overhead, CPU utilization, I/O efficiency, or hit rate. Not only will we compare the existing approaches proposed in the literature, but we shall also discuss new ideas, including some OS-related general solutions that had not attracted much attention previously. Since the traditional buffer pool management in disk-based databases has been shown to have very poor performance [23], [26], we shall ignore it in our study due to the space limitation.

Because of the diversity in focus and implementations, we are unable to fairly compare the performance of the entire systems. Instead, we will only focus on different "Anti-Caching" approaches. In order to eliminate the interference introduced by unrelated factors (e.g., the transaction manager, implementation choices, product orientation, etc.), we have implemented the various approaches on a uniform platform, namely Memcached [33], a light-weight in-memory key-value store with LRU eviction. We use the latest stable version 1.4.21 of Memcached in all the experiments described here.

We modified Memcached's eviction behavior so that it will put the evicted data onto disk rather than simply drop the data as in the original Memcached. The implementations of the different schemes were based on published papers, and source code provided by the authors when available. For the kernel-related approaches, we use Linux kernel 3.8.0 in the experiments. We shall detail our implementations and configurations in respective Sections IV-B, IV-C, IV-D, IV-E.

A. Environment Setup and Workload

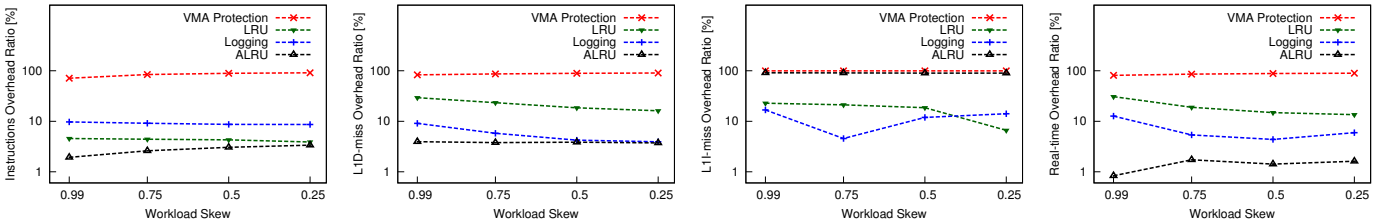
Our experiments are executed on a single x86_64 Ubuntu machine running Linux 3.8.0 kernel. The system consists of a dual-socket Intel Xeon E5-2603 Processor, 8GB of DDR3 RAM at 1066MHz and a commodity hard disk of 1TB. One socket connects 4 cores, which share a 10M L3 Cache, and each core has independent 32K L1 instruction cache, 32K L1 data cache and 256K L2 cache.

For all the experiments in this paper, we used the YCSB Benchmark 0.1.4 with Zipfian distributions [46]. The YCSB database in our experiments contains a single table with one million tuples, each of which has 10 columns. Each column consists of 100 bytes of randomly generated string data. We vary the skew factor of the request distribution from 0.99 to 0.25, and also ratios between available memory and data from 3/4, 1/2 to 1/4, to experiment with different hot data ratios and memory shortage scenarios. We generate enough workload requests to avoid the possibility that only a small percentage of tuples are accessed. We will only show the results for the "Read-Only" workload due to the space limitation. The other workloads exhibited similar results.

We remove most of the overhead from Memcached [33] in the experiments in order to only show the impacts of the "Anti-Caching" approach, and fine-tune each approach to get its best result. For example, to remove the network influence in the client-server architecture of Memcached, we executed the YCSB benchmark directly inside the server, rather than

TABLE II: Memory Overhead for Access Tracking

Methods	LRU	ALRU	Access Trace Logging	Page Table	VMA Protection
Memory Overhead Ratio	$\frac{8}{8+Size_{tuple}}$	$\frac{22/8}{22/8+Size_{tuple}}$	$\frac{C}{C+Count_{tuple}\times Size_{tuple}}$	$<<\frac{8}{8+Size_{page}}$	$<\frac{176}{176+Size_{vma}}$



(a) Instruction Overhead (b) L1-Data Cache Miss Overhead (c) L1-Instruction Cache Miss Overhead (d) Real Time Overhead

Fig. 2: CPU Overhead for Access Tracking

issue the requests through the network. In addition, we used *cgroup* to restrict the available physical memory to use for the benchmarked system so as to generate memory-constrained scenarios. All the experiments were repeated for three times, but we did not notice much variation in the results.

B. Access Tracking

By "access tracking", we mean the methods used to detect and record the data access behavior during the system execution. This information is used for deciding which data should be evicted. Fine-grained tracking may enable more accurate decision in the eviction phase, but it comes at a cost of higher memory and CPU overhead. In this subsection, we will investigate access tracking in terms of these overheads.

The access tracking methods we investigated are:

- LRU/ALRU. This is an intrusive strategy that requires modification of the applications. It generally involves adding customized access tracking code for each data access in the application. Here, we discuss two variants of LRU – exact LRU used by H-Store anti-caching [26] and Memcached [33] (LRU), and approximate LRU via aging used by Redis [34] (ALRU).
- Access trace logging. Instead of maintaining an in-memory LRU list, this strategy logs each access behavior for offline analysis that classifies data as hot or cold [30], [37]. The separation between normal execution and data classification reduces the interference to the system performance.
- Accessed and dirty flags in page table. There are two important flags in the page table entry that indicate the access status of a virtual page. These flags are utilized by the OS VMM for paging management. The "accessed" flag is set whenever there is an access operation on that page (read/write), while "dirty" flag is set when the page has been written to. These flags are set automatically by MMU whenever there is an access, and cleared periodically by the OS or kernel-assisted software so as to record the latest access information within a time slice [47], [38].
- VMA protection. Another mechanism that can be used to track the access behavior is to make use of virtual memory area (VMA) protection to restrict access so that when access does occur a segmentation fault will be triggered and escalated to the user [38]. For example, we can set the permission of an area to read-only (via system call *mprotect* or *mmap*) [3], in order to monitor write accesses. The available permission flags include `PROT_NONE` (i.e., cannot be accessed at all), `PROT_READ` (i.e., can be read), `PROT_WRITE` (i.e., can be modified), and `PROT_EXEC` (i.e., can be executed). An invalid access will trigger a segmentation fault, which can be caught by a registered page fault handler.

We shall analyze these access tracking approaches in terms of both memory and performance overhead, as it normally happens on the critical path of system execution, whose efficiency will definitely affect the overall performance.

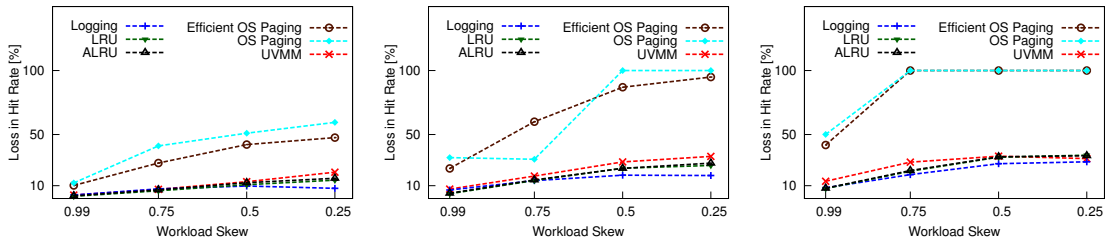
1) Memory overhead: Table II shows the memory overhead ratio for each method, where memory overhead ratio is defined as follows:

$$Overhead_{ratio} = \frac{R_{extra}}{R_{extra} + R_{useful}} \quad (1)$$

Here, R_{extra} denotes the extra resource (i.e., memory or CPU) used exclusively for "Anti-Caching" (i.e., extra memory used for access tracking in this context), and R_{useful} denotes the resource used for useful work (i.e., memory used to store the data in this context). We also use Equation 1 for the overhead of other components in the subsequent analysis.

We assume that we are using a 64-bit Linux environment as described in Section IV-A, i.e., the page size $Size_{page}$ is 4 KB, a pointer is 8 bytes, a page table entry is 8 bytes, and a VMA structure is 176 bytes. Furthermore, two pointers (i.e., 8 bytes) [4] are added in each tuple to act as the doubly-linked LRU list. Just like Redis [34], 22 bits are used for the age field, and the constant C is the size of the buffer used for logging. All the

[3]The size of a VMA area should be a multiple of the OS page size.
[4]We use 4-byte offset instead of 8-byte virtual address, as proposed by [26].



(a) Loss in Hit Rate ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{3}{4}$) (b) Loss in Hit Rate ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{1}{2}$) (c) Loss in Hit Rate ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{1}{4}$)

Fig. 3: Hit Rate for Different Eviction Strategies

quantities are in units of bytes. The values shown for "Page Table" and "VMA Protection" methods are the upper bounds, as access tracking may not necessarily create an extra page table entry/VMA structure. Such a structure may already exist for address mapping in normal execution. For instance, the VMA properties of neighbor address areas may be identical, making duplicates unnecessary.

From Table II, we can make the following observations:

- Aging-based LRU (ALRU) consumes less memory than exact LRU, especially for small tuple size. Trace logging maintains a buffer of a fixed size, so that the memory overhead is usually negligible.
- If the average tuple size is less than 4 KB for doubly-linked LRU list, or 1 KB for ALRU, their memory overheads are much higher than that of page-table-based tracking. More importantly, page-table-based tracking often comes for free, as it may be already used for address translation.
- In general, VMA protection is not cheap in terms of memory overhead if the permission change creates a large number of distinct VMAs. Thus, if this method is adopted, the granularity of VMA should be large enough to offset its memory overhead. But for a tuple size that is less than 180 bytes, it is still cheaper than LRU list even if the granularity is the same as page table (i.e., 4 KB). Furthermore, it is also probable that multiple continuous VMAs share the same properties (e.g., permissions, associated operations) and so a single VMA structure can cover multiple areas.

2) CPU overhead: We also measured the CPU performance penalty incurred by the access tracking process of the different methods. In particular, we shall report the overhead of instructions, Level 1 data cache (L1D) and instruction cache (L1I) misses, and real time (delay) incurred. As the overhead caused by page table update is experienced by any process running on top of the OS, it is unavoidable for all the methods. Hence, we will not consider that overhead here. We set the timer update interval to be 1 microsecond as was used by Redis [34], and the VMA flag reset interval as 1 second. The results reported in Figure 2 show that:

- The VMA protection method incurs a high overhead of more than 80%. Thus it is not a suitable solution for normal access tracking.
- The aging-based LRU (ALRU) is quite light-weight in terms of both instructions used and L1 data cache misses incurred, but it accounts for 90% of L1 instruction cache misses. This is because, in order to remove the overhead of obtaining the time (say, using `gettimeofday`) for every access, the time is only updated periodically. This is achieved by a timer event/signal – involving a system call that would result in a context switch (privilege mode), thus causing a high instruction cache miss.
- LRU list and access logging both have relatively low overheads that are nonetheless still significant. Specifically, LRU list update is likely to cause a high data cache miss rate, as the neighbor linked tuples are not likely to reside in the same cache line.

C. Eviction Strategy

In this subsection, we shall analyze the different eviction strategies, including tuple-level exact LRU (LRU) and aging-based LRU (ALRU), offline classification of hot or cold data (Logging) [31], Linux OS PFRA Paging mechanism (OS Paging) [42], and Efficient OS Paging [37]. We tested the hit rates of different strategies using the YCSB benchmark [46] with different skew ratios.

1) Optimal hit rate: In order to put the hit rate values in more perspective, we compare the values with an upper bound – the optimal hit rate under the same memory constraint, but with priori knowledge of all the data access traces. Firstly, we give the formal definition of the optimal hit rate as follows.

Definition Given a memory constraint $C_m = K(Objects)$ [5], and a complete n access traces $T = \{O_{t_0}, O_{t_2}, \ldots, O_{t_{n-1}}\}$, where O_{t_i} means that at the ith access, object O_{t_i} is accessed, the strategy with the largest hit rate is the optimal strategy, and its hit rate is the optimal hit rate. Formally, a strategy can be represented as a sequence of states $S = \{S_0, S_1, \ldots, S_{n-1}\}$, where $S_i = \{O_{i_0}, O_{i_1}, \ldots, O_{i_{(K-1)}}\}$, denotes the current set of objects in memory, and $|\{O_j \,|\, O_j \in S_i \, and \, O_j \notin S_{i+1}\}| \leq 1$. The latter means that only one object is fetched and evicted at any one time if needed. So the optimal strategy can be expressed as

$$S_{optimal} = \underset{S}{arg\max} \frac{\sum_{i \in \{0,\ldots,n-1\}} \mathbf{1}_{O_{t_i} \in S_i}}{n},$$

[5] We use the number of objects as the unit of memory constraint to simply the definition of the optimal hit rate.

TABLE III: Memory Overhead for Book-keeping

Methods	Evicted Table with Index	Bloom Filter	Page Table	VMA Protection
Memory Overhead Ratio	$\frac{20}{20+Size_{tuple}}$	$\frac{10/8}{10/8+Size_{tuple}}$	$< \frac{8}{8+Size_{page}}$	$< \frac{176}{176+Size_{vma}}$

given the complete knowledge of $T = \{O_{t_0}, O_{t_2}, \ldots, O_{t_{n-1}}\}$. The optimal hit rate is therefore

$$HR_{optimal} = \max_{S} \frac{\sum_{i \in \{0,\ldots,n-1\}} \mathbf{1}_{O_{t_i} \in S_i}}{n}.$$

2) Hit rate results: The loss in hit rate compared to the optimal hit rate under different scenarios (i.e., different ratios between memory size (**M**) and data size (**D**), different skew factors) are shown in Figure 3, from which, we can see that:

- The kernel-based eviction approaches suffer from poor eviction accuracy, which is mainly attributed by its less knowledge of user data structure. Even with a re-organization phase by users (i.e., Efficient OS Paging), the hit rate is not improved much, especially when the workload is less skewed and the available memory is more constrained.
- Access-logging-based offline classification has a good hit rate for all the scenarios, with a bit increase of the loss in hit rate with decreased workload skews. However, its disadvantage is that it cannot respond to workload changes quickly because there is a large latency from access logging, classification, to data migration.
- LRU and aging-based LRU (ALRU) have a relatively good hit rate. This is due to the fact that the eviction decision is based on a fine-grained tuple-level access tracking, and is adapted dynamically.

D. Book-keeping

"Anti-Caching" data layout can be abstracted as in-memory data, on-disk data, and book-keeping meta data that shows whether a tuple is in memory or on disk, which, however, is another source of memory overhead. Extra operations are also involved to update them on each data eviction so that the storage status change for the tuples to be evicted can be effected. The existing methods used for book-keeping are summarized as follows:

- Index update with in-memory eviction table, as is used by H-Store anti-caching [26].
- Separate stores for in-memory and on-disk data with access filters, as is used by Hekaton Siberia [30].
- Page table present flag used by the OS VMM [42].
- VMA protection, which is similar to the methods used in access tracking in Section IV-B. That is, we can change the protection of a VMA to `PROT_NONE`, causing accesses to this area to be trapped by a registered handler. It then fetches the data from disk according to a customized VMA table.

We shall only analyze the memory overhead in this section since the book-keeping performance overhead highly depends on the other components such as eviction strategy. Furthermore, it only happens when there is disk I/O, not in the primary execution path that dominates execution. The overall performance impact will be discussed in Section IV-F.

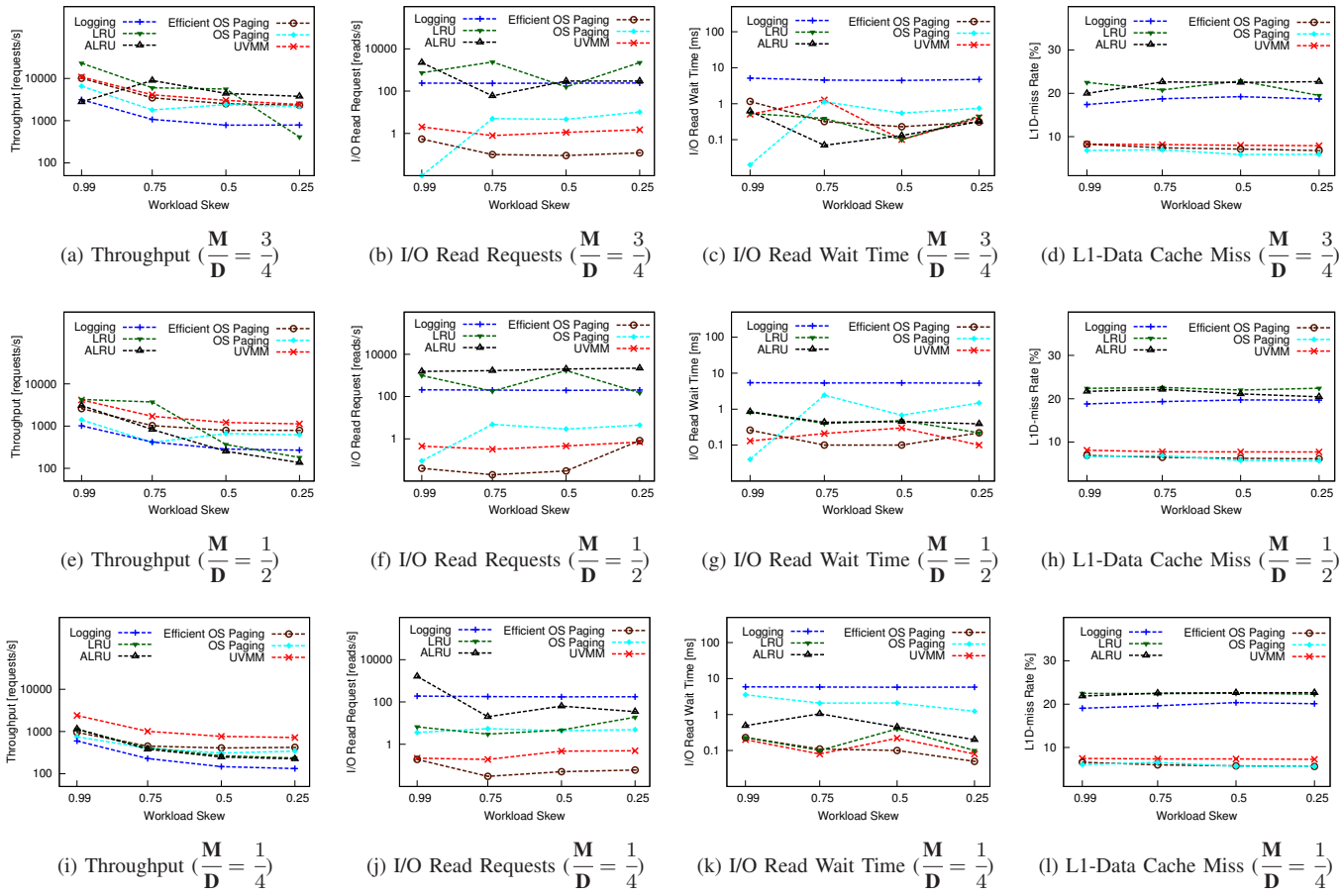
1) Memory overhead: The memory overhead ratios for the different methods are shown in Table III. We used Equation 1 as the definition of this ratio except that R_{useful} here refers to the size of memory saved by evicting some data (i.e., the total size of evicted data), and R_{extra} refers to the meta data used for the book-keeping of the status of evicted data. We used a memory-efficient representation for the evicted table and index, i.e., 8 bytes for key, 4 bytes for offset-based pointer, and 8 bytes for disk address (4-byte block ID and 4-byte offset as in [26]). Furthermore, we assumed a 1% false positive for the Bloom filter, which requires about 10 bits per tuple in general [48]. Table III yields the following insights:

- The book-keeping method using eviction table with index has a higher memory overhead. In the worst case, when the tuple size is small (e.g., 20 bytes), the memory saved by evicting the data is totally consumed by the book-keeping overhead.
- Access filter is a quite space-efficient method in general, where the overhead can be as little as 10 bits per tuple, however, at the price of extra cost penalty, caused by false positive accesses to the disk.
- The above observations make the page table and VMA protection methods more attractive since the same structures are used for multiple purposes (i.e., access tracking, book-keeping, and address translation).

E. Data Swapping

The efficiency of disk I/O plays an important role in the performance of "Anti-Caching", but we must not re-introduce the I/O bottleneck. The following methods for data swapping were evaluated:

- User-defined block-level swapping used in H-Store anti-caching [26]. This strategy utilizes the block device efficiently as OS Paging, especially in a memory-constrained environment without much free memory for the file system cache. However, the latency may be longer, and I/O traffic may be heavier as a single tuple read can only be serviced by a full block read. It wastes I/O since unnecessary data may be read back.
- Tuple-level migration used in Hekaton Siberia [30]. In particular, a tuple is the basic unit of swapping between hot and cold store, where the cold store is only required to provide basic storage functionality.
- Page-level swapping used in OS Paging [42]. As the minimum unit of data I/O is a page, it takes advantage of the block I/O property. Besides, a separate swap



(a) Throughput ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{3}{4}$) (b) I/O Read Requests ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{3}{4}$) (c) I/O Read Wait Time ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{3}{4}$) (d) L1-Data Cache Miss ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{3}{4}$)

(e) Throughput ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{1}{2}$) (f) I/O Read Requests ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{1}{2}$) (g) I/O Read Wait Time ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{1}{2}$) (h) L1-Data Cache Miss ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{1}{2}$)

(i) Throughput ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{1}{4}$) (j) I/O Read Requests ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{1}{4}$) (k) I/O Read Wait Time ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{1}{4}$) (l) L1-Data Cache Miss ($\frac{\mathbf{M}}{\mathbf{D}} = \frac{1}{4}$)

Fig. 4: CPU and I/O Performance

partition can also be utilized for data swapping, thus bypassing the file system overhead.

We will analyze these three methods for data swapping in terms of its I/O efficiency and CPU overhead using the YCSB benchmark, with combination of their respective strategies for other components in Section IV-F.

F. Performance – An Apple-to-Apple Comparison

In this section, we shall combine all the four components together in order to investigate how they affect each other and impact the overall performance (i.e., throughput). In order to have a fair comparison, we translate the memory overhead to a factor of the available memory that can be used, by provisioning each approach with the same size of physical memory. Therefore, approaches that have higher memory overheads will have less memory to manage for its core data. By doing so, memory overhead will have an impact on runtime performance.

Figure 4 shows the throughput, and also the CPU and I/O performance for the approaches we investigated, namely, H-Store anti-caching [26] (denoted as LRU), Redis aging-based LRU [34] (denoted as ALRU), Hekaton Siberia [30] (denoted as Logging), OS Paging [36], and Efficient OS Paging [37]. We also added our proposed approach in the figure, which will be elaborated in Section VI.

From Figure 4, we can conclude the following findings:

- Surprisingly, when there is a shortage of memory, the kernel-space approaches perform quite well, even though its hit rate is worse than user-space approaches' as shown in Figure 3. The main reason is that applications handle I/Os worse than the OS, which has a better I/O scheduler. Further, the user-space approaches experience longer latency because of the limited file system cache available. This can be seen from the I/O read requests per second and their average read latency.

- The LRU and aging-based LRU (ALRU) perform well when the workload is more skewed and the available memory is larger, as shown in Figure 4a and 4e, while their performance degrades greatly when the workload skew decreases and available memory is reduced. The poor performance is mainly caused by its higher memory overhead (resulting in less available memory for core data) and the interference between read and write I/O traffic caused by a larger I/O unit (i.e., 1M block size in the experiments).

- Even though logging-based approach (i.e., offline classification) has the highest hit rate as shown in Figure 3, it does not perform well in terms of throughput. This can be attributed by its tuple-level swapping strategy, causing higher I/O traffic and latency.

- Efficient OS Paging improves the default OS Paging significantly, resulting in a higher throughput and lower I/O traffic and latency, which is mainly contributed by injecting more data distribution information to OS Paging, enabling it make better eviction decision.
- The disparity of L1 data cache miss rates for user-space and kernel-space approaches is large, which shows the advantage kernel-space approaches have in terms of CPU utilization.

V. Implications on Designing a Good "Anti-Caching" Approach

A. Discussion with regards to other considerations

In addition to performance, there are other aspects that are also important to a system, which will be discussed bellow.

1) Generality/Usability: A general approach can be easily used in a common way by most systems without extensive modifications, while ad hoc approaches designed for specific systems cannot be easily adapted to other systems, thus involving a lot of repeated implementation. However, ad hoc approaches are able to adopt some application-specific optimizations, such as non-blocking transactions [26], and can operate on a finer-grained level (e.g., tuple-level) with more application semantics. In particular, OS Paging and caching are general enough to be applied in most systems naturally, while H-Store anti-caching [26] and Hekaton Siberia are designed for specific systems, i.e., H-Store and Microsoft Hekaton.

2) Responsiveness: A memory or performance overhead may have to be traded-off for a more responsive "Anti-Caching" approach, such as LRU, OS Paging, while offline approaches are normally based on the assumption that the workload is not often changing, and there must be a relative long delay for updating the classification result. Thus an online approach is more attractive for applications with an unpredicted workload ahead, but offline approaches usually incur less memory overhead.

B. Implications

Our analysis of state-of-art "Anti-Caching" approaches provides a set of insights on designing a good "Anti-Caching" approach that can efficiently handle the Big Data problem.

- In general, kernel-space approaches can take advantage of hardware-assisted mechanisms, incur less memory overhead, and embrace I/O efficiency. In contrast, user-space approaches are usually equipped with more semantic information, and finer operation granularity, thus enabling a more accurate eviction strategy.
- Hit rate is fundamental to determine a good "Anti-Caching" approach, which is difficult to achieve by only depending on the semantic-unaware access tracking methods (e.g., page table).
- I/O efficiency is of great importance, especially in a memory-constrained environment without much file system cache available. It can degrade the overall performance to a great extent if the I/O is not utilized efficiently.
- Higher memory overhead is detrimental to the performance, as it will be finally translated into performance degradation when the memory is not sufficient.

VI. Towards a General Approach with Efficient Utilization of Hardware

Based on the insights we have obtained, we shall now sketch a general user-space approach that not only has the flexibility of user-space implementation, but also the efficiency of OS VMM strategies, which we shall call as a *user-space virtual memory management* (UVMM). An important design goal of UVMM is to make it general enough to be easily applied to most in-memory database systems without much additional implementation work. We first describe the design principles of UVMM and then present the implementation details. Finally, we evaluate its performance.

A. Design Principles

As shown in Section IV and V, each component of "Anti-Caching" can potentially become a bottleneck in an in-memory database system. Rather than an ad hoc approach, having investigated the issues involved, we shall now propose a general yet efficient "Anti-Caching" approach based on the following design principles:

1) No indirection: As shown in [23], [49], the indirection of logical pointers (e.g., page ID and offset) used in traditional buffer pool management is the source of significant overhead – more than 30% of total execution time is spent on the extra indirection layer. Therefore, we should use real pointers for data access that utilize the hardware-assisted virtual address translation directly.

2) Non-intrusiveness: A general solution that can be easily applied without extensive modifications to the existing system is highly desirable. To this end, we believe that an API-based approach is most suitable. This will also enable backward compatibility, or transparent upgrading of existing applications.

3) Flexibility: We should provide a flexible list of options for applications that differ in their levels of intrusiveness. For example, applications can use an extended version of *malloc* with a preferred eviction strategy as an optional extension parameter. We should also provide several other APIs (e.g., access logging) for the application to disclose more semantics and hence improve the accuracy of the eviction. Without information on the semantics, access tracking could only be based upon the granularity of *malloc*-ed object, page and VMA. In addition, the APIs are implemented in the user-space, with a kernel module to assist privileged access to certain resources (e.g., page table). This reduces the communication need of applications while allowing them to take advantage of hardware assistance.

4) Reduced CPU overhead for normal operations: The "Anti-Caching"-related overhead should be minimized. To track the access behavior, we propose the use of a combination of page table information and optional user supplied access logging. The update of access information from page table is done in a fine-grained non-blocking manner so as to avoid interference with normal execution.

5) Reduced memory overhead: In order to lower the memory overhead ratio, we propose keeping the meta data at the page level. To avoid inaccurate replacement decisions caused by coarse-grained knowledge of data access behavior, we will also require additional fine-grained access distribution information. Such information is acquired using optional user logging and a customized kernel page table access module.

B. Implementation

We shall elaborate on our implementation of UVMM in terms of the four aspects, namely, access tracking, eviction strategy, book-keeping and data swapping. Due to space limitation, we are only able to describe the implementation techniques and strategies that we used at a fairly high level.

1) Access tracking: We implemented a combination of access tracking methods, namely, VMA protection, page table, *malloc*-injection and access logging, in order to make it flexible enough to be used by upper-layer applications that may have different concerns. Because of its high overhead, the VMA protection tracking method (shown in Figure 2) is only used when we have to fetch data from disk. With this strategy, the overhead of a segmentation fault is easily offset by that of a disk read, and the access tracking and book-keeping are completed at the same time. In addition, *malloc* is also used to indicate the access behavior at the granularity of *malloc*-ed size. We can update our own access information on the granularity of page by periodically querying the page table in a fine-grained asynchronous way. Optionally, users can choose to log data access at the granularity of tuples. This however does not require the writing of data to a log file on disk. Instead, it only switches some bits in the memory, which can be done quite efficiently.

2) Eviction strategy: We implemented a variety of eviction strategies for use in order to meet different requirements of various workloads since no single eviction strategy is optimal for all use cases. The eviction strategies available currently include aging-based LRU, WSCLOCK, FIFO, RANDOM, and optimized WSCLOCK with consideration of user-provided access logging. In particular, for the optimized WSCLOCK algorithm, we not only consider the accessed and dirty flags as is the case for WSCLOCK, but also consider the distribution of data access within one VMA. This improves the selection accuracy of eviction candidates.

3) Book-keeping: We use the VMA protection method with a larger protection area size for book-keeping in order to reduce the memory overhead. Hardware virtualization [50] is a potential technique for reducing the segmentation fault overhead. It also allows for access to privileged resources such as page table directly without the involvement of a kernel module, making page-table-based book-keeping method available in the user-space.

4) Data swapping: In our implementation, data swapping is conducted in the unit of VMA. To reduce the possibility of disk I/O wastage, we implemented a combination of fine-grained access tracking methods described before to make more accurate eviction decision and increase our hit rate. In addition, we also used a fast compression technique – lz4 [51] to further reduce the I/O traffic at the price of a little CPU overhead. We believe this is a reasonable tradeoff. Besides, asynchronous I/O is used to make I/O operations non-blocking, removing the I/O bottleneck in the critical path. We are also experimenting with a strategy where we first compress the data and put it in a fixed memory region. Only when that region is full do we perform the actual disk I/O.

C. Performance Results and Analysis

We evaluate the performance of UVMM against the other approaches we investigated. The results are shown in Figure 3 and Figure 4 where our approach is labeled as "UVMM". From Figure 3, we can see that, with the more fine-grained information we obtain through the general APIs (e.g., *malloc*, access logging), the accuracy of our eviction strategy is improved significantly, thus leading to a better throughput and more efficient utilization of CPU and I/O (Figure 4).

Specifically, the reasons for the good performance can be summarized as follows.

- The access tracking is light-weight and incurs only a small overhead in execution.
- We have more semantics information provided by the user applications, such as data object size and fine-grained access distribution within a page, which enables a better eviction strategy.
- The kernel-supported VMA protection mechanism provides us an efficient book-keeping method without much memory overhead.
- Compression significantly reduces I/O traffic (more than 20%), and kernel-supported asynchronous I/O takes advantage of the efficient kernel I/O scheduler.

VII. Conclusions

The "anti-caching" approach enables in-memory database systems to handle big data. In this paper, we conducted an in-depth study on the state-of-the-art "anti-caching" approaches that are available in user- and kernel-spaces by considering both CPU and I/O performance, and their consequential runtime system throughput. We found that user- and kernel-space approaches exhibit strengths in different areas. More application semantics information is available to user-space approaches which also have finer operation granularity. This enables a more accurate eviction strategy. Kernel-space approaches, on the other hand, can directly use hardware (CPU and I/O) assistance, and are more likely to provide good resource utilization. It is therefore promising to combine these strengths, and we sketched a general user-space virtual memory management approach that efficiently utilizes hardware through kernal support. Our experiments gave evidence of the potential of such a holistic approach. We hope this study will contribute towards the future realization of this combined approach in actual in-memory database systems.

Acknowledgment

This work was supported by the National Research Foundation, Prime Minister's Office, Singapore, under its Competitive Research Programme (CRP Award No. NRF-CRP8-2011-08). We would also like to thank Bogdan Marius Tudor for his insightful suggestions and comments.

REFERENCES

[1] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis *et al.*, "The case for ramclouds: Scalable high-performance storage entirely in dram," *Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2010.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes *et al.*, "Spanner: Google's globally-distributed database," in *OSDI '12*, 2012, pp. 251–264.

[3] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm *et al.*, "Asterixdb: A scalable, open source BDMS," in *PVLDB '14*, 2014, pp. 841–852.

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI '04*, 2004, pp. 137–150.

[5] D. Jiang, G. Chen, B. C. Ooi, K.-L. Tan, and S. Wu, "epic: an extensible and scalable system for processing big data," in *PVLDB '14*, 2014, pp. 541–552.

[6] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert *et al.*, "Pregel: A system for large-scale graph processing," in *SIGMOD '10*, 2010, pp. 135–146.

[7] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin *et al.*, "Distributed graphlab: A framework for machine learning and data mining in the cloud," in *PVLDB '12*, 2012, pp. 716–727.

[8] BackType and Twitter, "Storm: Distributed and fault-tolerant realtime computation," 2011. [Online]. Available: https://storm.incubator.apache.org/

[9] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *ICDMW '10*, 2010, pp. 170–177.

[10] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro *et al.*, "Implementation techniques for main memory database systems," in *SIGMOD '84*, 1984, pp. 1–8.

[11] T. J. Lehman and M. J. Carey, "A recovery algorithm for a high-performance memory-resident database system," in *SIGMOD '87*, 1987, pp. 104–117.

[12] M. H. Eich, "Mars: The design of a main memory database machine," in *The Kluwer International booktitle in Engineering and Computer Science*. Springer US, 1988, vol. 43, pp. 325–338.

[13] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 509–516, 1992.

[14] R. Kallman, H. Kimura, J. Natkins, A. Pavlo *et al.*, "H-store: A high-performance, distributed main memory transaction processing system," in *PVLDB '08*, 2008, pp. 1496–1499.

[15] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *ICDE '11*, 2011, pp. 195–206.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI '12*, 2012, pp. 15–28.

[17] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for dram-based storage," in *FAST '14*, 2014, pp. 1–16.

[18] V. Sikka, F. Färber, W. Lehner, S. K. Cha *et al.*, "Efficient transaction processing in sap hana database: The end of a column store myth," in *SIGMOD '12*, 2012, pp. 731–742.

[19] T. Lahiri, M.-A. Neimat, and S. Folkman, "Oracle timesten: An in-memory database for enterprise applications," *IEEE Data Engineering Bulletin*, vol. 36, no. 2, pp. 6–13, 2013.

[20] J. Lindström, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila, "Ibm soliddb: In-memory database optimized for extreme speed and availability," *IEEE Data Engineering Bulletin*, vol. 36, no. 2, pp. 14–20, 2013.

[21] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson *et al.*, "Hekaton: Sql server's memory-optimized oltp engine," in *SIGMOD '13*, 2013, pp. 1243–1254.

[22] D. Sidlauskas and C. S. Jensen, "Spatial joins in main memory: Implementation matters!" in *PVLDB '15*, 2014, pp. 97–100.

[23] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "Oltp through the looking glass, and what we found there," in *SIGMOD '08*, 2008, pp. 981–992.

[24] H. Zhang, B. M. Tudor, G. Chen, and B. C. Ooi, "Efficient in-memory data management: An analysis," in *PVLDB '14*, 2014, pp. 833–836.

[25] H. Zhang et al., "Memepic: Towards a database system architecture without system calls," National University of Singapore, Tech. Rep., 2014.

[26] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik, "Anti-caching: A new approach to database management system architecture," in *PVLDB '13*, 2013, pp. 1942–1953.

[27] A. Silberschatz, P. B. Galvin, G. Gagne, and A. Silberschatz, *Operating system concepts*, 1998, vol. 4.

[28] B. Jacob, S. W. Ng, and D. T. Wang, *Memory systems: cache, DRAM, disk*, 2010.

[29] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser *et al.*, "Instant loading for main memory databases," in *PVLDB '13*, 2013, pp. 1702–1713.

[30] A. Eldawy, J. J. Levandoski, and P. Larson, "Trekking through siberia: Managing cold data in a memory-optimized database," in *PVLDB '14*, 2014, pp. 931–942.

[31] J. J. Levandoski, P. Larson, and R. Stoica, "Identifying hot and cold data in main-memory databases," Washington, DC, USA: in *ICDE '13*, 2013, pp. 26–37.

[32] K. Alexiou, D. Kossmann, and P.-A. Larson, "Adaptive range filters for cold data: Avoiding trips to siberia," in *PVLDB '13*, 2013, pp. 1714–1725.

[33] B. Fitzpatrick and A. Vorobey, "Memcached: a distributed memory object caching system," 2003. [Online]. Available: http://memcached.org/

[34] S. Sanfilippo and P. Noordhuis, "Redis," 2009. [Online]. Available: http://redis.io

[35] W. Effelsberg and T. Haerder, "Principles of database buffer management," *ACM Transactions on Database Systems*, vol. 9, no. 4, pp. 560–595, 1984.

[36] A. S. Tanenbaum, *Modern operating systems*, 1992, vol. 2.

[37] R. Stoica and A. Ailamaki, "Enabling efficient os paging for main-memory oltp databases," in *DaMoN '13*, 2013, p. 7.

[38] F. Funke, A. Kemper, and T. Neumann, "Compacting transactional data in hybrid oltp&olap databases," in *PVLDB '12*, 2012, pp. 1424–1435.

[39] R. Bose and J. Frew, "Lineage retrieval for scientific data processing: A survey," *ACM Computing Surveys*, vol. 37, no. 1, pp. 1–28, 2005.

[40] M. Rajashekhar and Y. Yue, "Twemcache: Twitter memcached," 2012. [Online]. Available: https://github.com/twitter/twemcache

[41] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski *et al.*, "Scaling memcache at facebook," in *NSDI '13*, 2013, pp. 385–398.

[42] M. Gorman, *Understanding the Linux virtual memory manager*, 2004.

[43] MongoDB Inc., "Mongodb," 2009. [Online]. Available: http://www.mongodb.org/

[44] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution," in *CIDR '05*, 2005, pp. 225–237.

[45] H. Chu, "Mdb: A memory-mapped database and backend for openldap," *The 3rd LDAP International Conference*, 2011.

[46] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," New York, NY, USA: in *SoCC '10*, 2010, pp. 143–154.

[47] D. P. Bovet and M. Cesati, *Understanding the Linux kernel*, 2005.

[48] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," London, UK, UK: in *ESA' 06*, 2006, pp. 684–695.

[49] G. Graefe, H. Volos, H. Kimura, H. Kuno *et al.*, "In-memory performance for big data," in *PVLDB '15*, 2014.

[50] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei *et al.*, "Dune: Safe user-level access to privileged cpu features," in *OSDI '12*, 2012, pp. 335–348.

[51] Y. Collet, "Lz4: Extremely fast compression algorithm," 2013. [Online]. Available: https://code.google.com/p/lz4/