A Persistent Memory-Aware Buffer Pool Manager Simulator for Multi-Tenant Cloud Databases

Taras Basiuk
School of Computer Science
University of Oklahoma
Norman, USA
taras.basiuk@ou.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, Oklahoma, USA
ggruenwald@ou.edu

Laurent d'Orazio
CNRS IRISA
Rennes 1 University
Lannion, France
laurent.dorazio@univ-rennes1.fr

Eleazar Leal
Dept. of Computer Science
University of Minnesota Duluth
Duluth, USA
eleal@d.umn.edu

***Abstract*—Non-Volatile Memory (NVM) is a promising development for Database Management Systems (DBMS), offering abundant and fast storage to complement traditional disk and main memory architectures. NVM introduces additional data migration possibilities to the traditional buffer pool (BP) managers used by the DBMS. Hence, the efficient use of this new technology requires a re-design of the BP manager. For the cloud Database-as-a-Service products, this need for a re-design is further complicated by the traditional cloud providers' goal to minimize the Service Level Agreement (SLA) violation penalties paid to their tenants. Unfortunately, current research in the area does not provide a comprehensive picture of the components constituting a multi-tenant persistent memory aware BP manager for a cloud database that makes use of NVM. Furthermore, researchers lack the software tools needed to quickly prototype and estimate the effectiveness of novel data management policies guiding those components. In this paper, we attempt to remedy both issues, first, by proposing a generalized framework that defines the purpose and the abstract interfaces of various multi-tenant persistent memory-aware BP manager components, and second, by developing and demonstrating a simulator algorithm that is shown to aid in quick testing of different implementations of those BP manager components.**

Keywords—database systems, nonvolatile memory, buffer storage, cloud computing, simulation

I. INTRODUCTION

Modern cloud software providers, such as Amazon AWS, Microsoft Azure, and Google Cloud Services, offer their Database-as-a-Service (DBaaS) products via an agreement with their clients (known as tenants), called Service Level Agreement (SLA). It includes a set of technical Service Level Objectives (SLOs) that the provider promises to satisfy (e.g., service uptime or query response latency) or be forced to pay an SLA violation penalty cost to the tenant. To avoid paying high infrastructure and SLA violation penalty costs, providers share the hardware resources they own (database servers) among many tenants.

Traditional disk-oriented database management systems (DBMS) deployed to the cloud use a two-tier data storage system consisting of a persistent disk drive and random-access memory (RAM). On the one hand, disk drives, such as either solid-state drives (SSD) or hard-disk drives (HDD), have a low cost per byte, are abundant, and have long latency. RAM, on the other hand, is volatile (data is lost upon device power loss), more expensive per byte, limited in size, but has shorter latency.

The database buffer pool (BP) manager is a software component of the DBMS responsible for migrating data from disk to memory (in the form of disk pages) for query processing, keeping frequently accessed data in memory, and migrating dirty data from memory back to disk for persistence. Given the significant data access performance difference between disk and main memory, the tenant's query processing SLA satisfaction level differs drastically when the relevant data is available in-memory versus when it is on-disk. Hence, BP management policies responsible for the choice of the data to be migrated to the BP memory (which is limited, expensive, and shared between tenants) and the choice of data to be evicted from it, are crucial for the overall number of SLA violations that the cloud database system produces while processing tenant queries.

A recent development in data storage technologies (dating back to the 2015 Intel and Micron announcement of the 3D XPoint technology [1]) has disrupted the traditional two-tier storage and BP manager architectures and designs. Non-volatile memory (NVM) is a novel family of data storage manufacturing technologies that promise and, to some extent, already delivers [1, 2] storage that is almost as fast as RAM, persistent, byte-addressable, and nearly as abundant and as cheap as disk.

NVM can serve as the third tier of storage, which can be logically placed between memory (RAM) and disk tiers [3, 4] (in terms of data access performance versus available space and price). This addition requires a redesign of existing approaches to optimal buffer pool resource management in the general case [1, 3, 4, 5, 6], and by extension, for a multi-tenant cloud database management system. However, as [1] points out, NVM-related DBMS research currently relies upon the ability to quickly build and evaluate the software component prototypes, which requires the assistance of various software and hardware tools. Unfortunately, such tools are not currently available to the researchers not affiliated with the NVM hardware manufacturers or with the proprietary DBMS vendors [1].

To help with the future NVM-related cloud DBMS research, we propose the *CPBPSim* algorithm (Cloud Persistent Memory-Aware Buffer Pool Simulator). The purpose of this simulator is to quickly estimate the SLA violation penalty caused by the operation of the BP in its given configuration of components

This work is partially supported by the National Science Foundation Award No. 1349285.

while processing a given sequence of data page accesses. However, in order for our algorithm to be truly useful for future research, we must make it easy to interact with. Otherwise, the researchers will spend too much of their time adhering to the requirements of our algorithm, which defeats the purpose of quick prototype evaluation. To address this challenge, we also contribute the *CPBPSim* framework, which we generalize from prior research related to components of NVM-aware and multi-tenant BP managers. The framework lists the components required for the BP manager simulation, their purpose, and the way the simulator algorithm will interact with them.

The rest of this paper is organized as follows: Section II presents the *CPBPSim* framework and discusses the prior research related to its components. Section III describes the *CPBPSim* algorithm. Section IV demonstrates the operation of the simulator using sample data sets and example implementations of the framework components. Finally, Section V provides conclusions and future research plans.

II. Proposed Simulator Framework

A. Framework Overview

The general operation of the *CPBPSim* framework is illustrated in Fig. 1. The centerpiece of the framework is the simulation algorithm, the details of which are described in Section III. It takes a sequence of data page access requests as an input, considering them one by one (1). For each such request, the algorithm retrieves the available page metadata (2), such as the storage tier it is currently located at and whether it is dirty (updated while on a volatile storage tier). Then, the algorithm will consult the data admission policy of the current storage tier on whether the page should be admitted to a different tier (3). If so, it will consult the data migration policy of the tenant the page belongs to, to determine the destination tier (4). If the destination tier lacks free space, its data eviction policy will be consulted to select a page that should be evicted (5). The data migration policy of the tenant whose page just got evicted will be consulted to find a destination for that page. Steps (5) and (4) may be repeated multiple times, forming a *data access chain*, until the final destination tier has free space without having to evict any page. If the data admission policy indicated (3) that the original page should stay where it is, the *data access chain* would only contain a single entry to serve access to the requested page. Then the *data access chain* will be processed by the *CPBPSim*, which

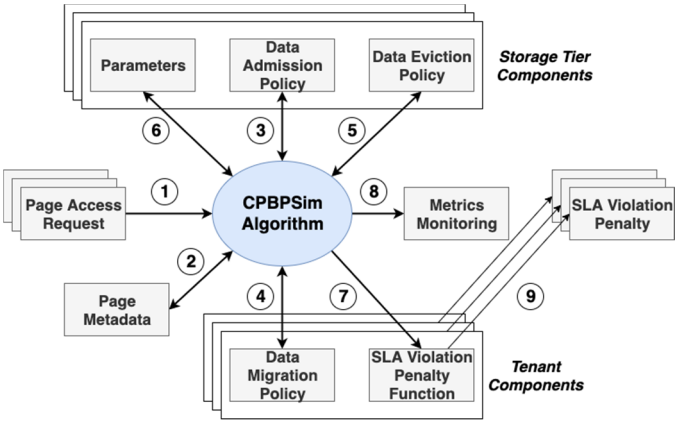

Fig. 1. CPBPSim Framework Diagram

involves updating the page metadata (2) with the new page location and the dirty/clean status and interacting with the storage tier parameters (6) to update the available free space and calculate the SLO costs incurred during the simulated reading and updating of the pages on different tiers (total latency, for example). The calculated SLO cost will be recorded in the SLA violation penalty function of the tenant who made the page access request (7). The SLO cost complemented by other metrics relevant for the simulator monitoring (for example, *data access chain* length and the initial storage tier of the requested page) will be next recorded in the metrics monitoring component (8). Once all data access requests are served, the SLA violation penalty function of every tenant is evaluated to map the recorded SLO costs to SLA penalties (9). Researchers using the simulator then also can inspect the collected metrics to guide their simulator configuration decisions for future simulations.

Next, we discuss the details of the four major components of the framework, Data Admission Policy, Data Migration Policy, Data Eviction Policy, and SLA Violation Penalty Functions, and derive their requirements from the prior research.

B. Data Admission Policy

The purpose of a Data Admission Policy is to detect when a recent access pattern to a page currently residing in a storage tier becomes "too hot" for that tier and should be admitted to a tier where it will be better served.

In traditional DBMSs, which only has the SSD and RAM storage tiers, there is no need for a Data Admission Policy because when a page of data is requested, it is either already in RAM or has to be copied there in order for the CPU to access it. With the addition of the NVM storage tier (which is also accessible by the CPU), there are now two choices when a page is first accessed: 1) to keep the page in NVM, or 2) to promote it to RAM and then service it from RAM, which could result in faster accesses for future references to that same page.

To the best of our knowledge, only in [5], the above data admission scenario was considered explicitly, and the authors developed a novel 2Q algorithm to address the challenge. In [4], data is served only from the RAM tier, so when a page of data resides in SSD, it is always copied in its entirety to RAM. However, when the page resides in NVM (after being evicted from RAM), the tuples required by the query processing belonging to that page are copied to RAM in the form of either cache-line-grained page (mostly empty page) or mini-page (page of smaller size). That is the only practical usage of the byte-addressability of NVM for BP management known to us to date. In [3], the authors mention that the data is "eventually" admitted from NVM to RAM, but do not explicitly describe the BP manager policy responsible for this eventuality.

Even though there is not much prior research on Data Admission Policies and the fact that existing approaches are not easily generalizable, we can require *CPBPSim* to support the *heterogeneous data admission policies* to be able to account for different bottlenecks in different storage tiers, and we define the pseudocode interface for such policies in Section III.

C. Data Migration Policy

The purpose of a Data Migration Policy is to decide upon admission or eviction of a data page from a particular storage

tier, the destination storage tier, for that page. The need for it emerged due to the addition of the third tier of storage, NVM.

The issue of data migration policies was addressed in [3], where the authors use probability-based data migration policies which encode the likelihood that a page will bypass NVM on its way to RAM upon the admission from SSD, or the likelihood to bypass NVM on its way to SSD upon eviction from RAM. The authors use simulated annealing to find the optimal values of the probabilities encoded in the policy for the given workload. The Data Migration Policy considered in [4] decides the destination storage tier for a page of data evicted from the RAM tier. It keeps track of recently evicted pages, and if the page in question is among them, it will be copied to NVM, instead of SSD.

In the general case, not only the source and the destination storage tiers (returned by a Data Migration Policy) might end up being the same, but also a case might occur when, for example, upon the eviction of a page from the NVM tier, the page gets migrated to the RAM tier. For a centralized single-user DBMS, considering such cases does not make sense (if the Data Eviction Policy says a page should be evicted from NVM, why would we migrate it back to NVM?). However, in the case of a multi-tenant DBMS, we speculate that the page access patterns of a minority of tenants (and the Data Migration Policies which are optimal for them) can conflict with the Data Admission and Eviction policies which work well for a majority of tenants. Hence, *CPBPSim* supports *heterogeneous data migration policies* (different for different tenants) and can resolve the *data access chains* of variable length. For example, the NVM-RAM-NVM-SSD chain means that when admitting a page from NVM to RAM, we have to evict another page from RAM to NVM, which requires evicting the third page from NVM to SSD. The data migration policy interface is given in Section III.

D. Data Eviction Policy

Also known as a Page Replacement Policy, the Data Eviction Policy is an integral part of a traditional BP manager. Its purpose is to select a victim data page to be removed from a storage tier, which lacks free space to host another data page. The decision is usually made based on the recorded history of accesses to specific data pages with the goal of keeping the pages which are more likely to be accessed in future (often referred to as hot pages) in the storage tier which can service the access quicker but is limited in size (traditionally RAM).

The addition of the NVM storage tier to the BP prompted the researchers to propose the following decisions regarding the data eviction policies. In [4], the authors use two separate instances of identical policies (one for the RAM tier and one for the NVM tier) based on the Clock algorithm [7]. For simplicity, they use identical policies but suggest that using different policies for different tiers might yield better results. In [5], an eviction policy based on the LRU algorithm is used to evict pages from the RAM tier to the NVM tier. It is hard to tell which policies are used in [3], as the authors say that the data is evicted "eventually" from the RAM and NVM buffers, and then, focus only on the data migration aspect.

To generalize the above approaches, we consider the support of the *heterogeneous data eviction policies* (different policies for different storage tiers) to be a requirement of the *CPBPSim* so it can allow for effectively working around different bottlenecks of different storage tiers (for example, available space is a bottleneck of RAM, while write endurance is a bottleneck of NVM [1]). In Section III, we define an interface for an abstract data eviction policy through which the *CPBPSim* algorithm interacts with specific implementations of the policy.

E. SLA Violation Penalty Functions

The purpose of the SLA violation penalty function is to map the level of the promised SLO parameter satisfaction to a penalty accrued by the cloud provider and paid to the tenant. There are three components at play here: the choice of the SLO parameter, the evaluation period, and the mapping function.

Existing industrial DBaaS products overwhelmingly offer simple service availability as their SLO[1,2], with only some offering more advanced SLOs, such as query latency, throughput, etc.[3] Research papers, in addition to the SLOs used in the industry, also consider resource utilization (CPU, RAM, Disk I/O) by tenants [8]. The only paper we are aware of related to multi-tenant BP management, which deals with SLA, uses the cache hit rate degradation as its SLO [9]. The addition of NVM to the BP manager suggests considering additional metrics, such as NVM space utilization, number of writes (as NVM has limited physical endurance compared to RAM [1]), and power consumption, as new candidate SLOs.

The SLA penalty evaluation period can be on a per-event basis (for example, each query processing time can incur some penalty), but usually is aggregated over some time period (for example, service availability for the past month[1,2,3] or 99th percentile of query response latency for the past hour[3]).

The function which maps the SLO parameter satisfaction level to the accrued penalty can be a simple step-based function (for example, the monthly service availability is less than 99%, 50% of subscription cost is credited back to the tenant; otherwise no penalty assigned), or it can be a more advanced piece-wise linear function [9]. In general, it must be able to provide a one-to-one SLO parameter to the SLA violation penalty value mapping. The SLA violation penalty function pseudocode interface used by the CPBPSim is described in Section III.

III. Proposed Simulator Algorithm

In this section, we first introduce the interfaces for the abstract components of the framework described in Section II, and then, provide the pseudocode of the *CPBPSim* algorithm.

A. Component Interfaces

We begin with the Data Eviction Policy. Our study of the common algorithms used in existing data eviction policies (LRU and Clock [7], and LRU-K [10]) allows us to assume that *CPBPSim* can effectively interact with an abstract data eviction policy via an interface defined in Algorithm 1. The simulator will initialize the policy (Line 1) once with the specific storage tier parameters (e.g., maximum available size and access latencies) and policy-specific configuration parameters (e.g., a value of K for the LRU-K policy). Each time a page is requested

[1] https://aws.amazon.com/rds/sla/

[2] https://cloud.google.com/sql/sla

[3] https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/

in the simulator, the record_access function (Line 4) will be called for each data eviction policy with the values of a current system timestamp, *pageID*, access type (e.g., read or write), and whether the page resides in the tier. The reason why each policy has to record page access, regardless of which storage tier is currently hosting the page, is because any tier might end up hosting the page, and any such policy might have to make an eviction decision regarding it. Furthermore, the information that makes a data page the right candidate for eviction from one storage tier does not necessarily make it the right candidate for eviction from another (think of the frequency of reads in RAM vs. the frequency of writes in NVM). When a particular tier runs out of free space to host another page, the evict method (Line 7) will be called on the policy to find a *pageID* to evict and replace it with the new page. Note that when a page of data is evicted from a storage tier, if it was modified, its state has to be persisted on some other storage tier (decided by the data migration policy described below) and cannot be thrown away. Finally, when a page is migrated from one tier to another, its residency with the policy must be updated (Line 11), so the page will not get evicted when it is not resident.

We define a very similar interface in Algorithm 2 for the *CPBPSim* to interact with an abstract data admission policy. The purpose and the signature of the init and record_access functions remain the same as for the Abstract Data Eviction policy. The purpose of the should_admit function (Line 7), however, is for the policy to decide whether the given *pageID* is "too hot" for its current storage tier and should be admitted to some other tier.

The two examples [3, 4] of the Data Migration Policies we mentioned in Section II allow us to define another interface in Algorithm 3, with which the *CPBPSim* interacts. The two functions playing the central role are dest_on_admit (Line 4) and dest_on_evict (Line 8), which decide the destination storage tier for the given *timestamp*, *pageID*, and *source* storage tier. Note that the init function (Line 1) only takes the policy configuration parameters as input but does not take any input related to the storage tier parameters. We consider the Data Admission and Eviction policies to be the BP manager components that are uniquely responsible for managing the contents of different storage tiers taking their various parameters into consideration.

Next, we generalize the SLA Violation Penalty Functions reviewed in Section II to an interface provided in Algorithm 4, with which *CPBPSim* will interact. The function is initialized (Line 1) with the type of SLO parameter, an evaluation period, and a mapping function. Periodically the simulator will call the record_slo function (Line 4) with the timestamped SLO satisfaction value. The eval_penalty function (Line 7) can be called to get the total penalty accrued by the cloud provider since a specific time. Generally, our simulator will be interested in the total penalty accrued during the simulation. However, some Data Eviction Policies, such as MT-LRU [9], may make their eviction decision based on the penalty accrued during the current evaluation period, the cutoff time is therefore needed.

The interface for an abstract metrics monitor is given in Algorithm 5, which allows the simulator to keep track of data access chains, SLO costs, origin tiers of requested pages, and similar metrics.

Algorithm 1 Abstract Data Eviction Policy

```
1.  function init(storage_params, config_params)
2.      # Policy initialization
3.  end function
4.  function record_access(timestamp, pageID, type, resident)
5.      # Internal state of the policy is updated
6.  end function
7.  function evict()
8.      # Decide which page to evict
9.      return pageID
10. end function
11. function update_residency(pageID, resident)
12.     # Update the residency flag of the page with this policy
13. end function
```

Algorithm 2 Abstract Data Admission Policy

```
1.  function init(storage_params, config_params)
2.      # Policy initialization
3.  end function
4.  function record_access(timestamp, pageID, type)
5.      # Internal state of the policy is updated
6.  end function
7.  function should_admit(pageID)
8.      # Decide whether to admit the page
9.      return True/False
10. end function
```

Algorithm 3 Abstract Data Migration Policy

```
1.  function init(config_params)
2.      # Policy initialization
3.  end function
4.  function dest_on_admit(timestamp, pageID, source)
5.      # Decide the destination tier
6.      return destination
7.  end function
8.  function dest_on_evict(timestamp, pageID, source)
9.      # Decide the destination tier
10.     return destination
11. end function
```

Algorithm 4 Abstract SLA Violation Penalty Function

```
1.  function init(slo_type, eval_period, mapping_func)
2.      # Penalty function initialization
3.  end function
4.  function rec_slo(timestamp, slo_val)
5.      # Record the SLO value
6.  end function
7.  function eval_penalty(from_time)
8.      # Calculate the penalty accrued since from_time
9.      return penalty
10. end function
```

Algorithm 5 Abstract Metrics Monitor

```
1.  function init(config_params)
2.      # Monitor initialization
3.  end function
4.  function record_metric(timestamp, tenant, tier, chain, slo_val)
5.      # Internal state of the monitor is updated
6.  end function
7.  function aggregate_metrics(from_time)
8.      # Aggregate the metrics since from_time
9.      return metrics
10. end function
```

B. Algorithm Implementation

Finally, consider Algorithm 6, which provides the specifics of *CPBPSim*. The inputs of the algorithm are (1) the components discussed in Section II; (2) *N* data page access requests, each consisting of, *t* - timestamp, *p* - data page ID, *ten* - tenant ID, and *type* - page access type (reads and updates); (3) *prms* - storage tier parameters of every considered storage tier, consisting of, free space available, *cpu_acc* flag (deciding whether the tier can be accessed by the CPU and hence serve the data access requests), and various SLO contributions (costs) for

various access types (for example, *prms*[RAM][R][latency] = 1ns means that read access to RAM incurs 1ns latency SLO cost); (4) *meta* – metadata the simulator collects about the data pages, consists of the current storage *tier* of every page and its *dirty* flag; and (5) *mon* – metrics monitor.

The algorithm works by sequentially considering every page access request (Line 2). First, we check whether the requested page is in the storage tier that can be accessed by the CPU and should not be admitted to some other tier (Line 5). If so, the only entry in the data access chain will be to serve the request (Line 6) without actually migrating any other pages. If either of those conditions is not true, we free up space in the current tier, update the page residency with its current Data Eviction Policy (Lines 8 and 9), and invoke the Data Migration Policy to find a new tier accessible by the CPU (Lines 11 to 13). Then, we check whether the new tier has enough free space. If so, we add an entry to the data access chain to serve the request from the new tier (Line 18) and decrement the amount of free space (Line 20). If there is not enough space, we invoke the Data Eviction Policy to find a victim page (Line 23); if it is dirty, we check to which tier we can migrate it (Line 25) and prepend a "copy" access type to the data access chain. We continue doing so until we evict a page that is not dirty, or we migrate it to the tier which has enough space. After that happens, we start processing the data access chain (Line 34). For each entry in the chain, we update the SLO contribution for the corresponding access type (Line 36). If the access type is "copy", it will be counted as an "update" access on this tier, but we will also update the SLO value with the cost or "read" from the source tier of the copy (Line 39). We also update the meta with the updated tiers of the pages (Line 41), clear the dirty flag of the pages migrated to the SSD tier (Line 43), and set the dirty flags for the pages updated while not in SSD (Line 46). Finally, for every page access request, we record it for the Data Admission and Eviction Policies of every storage tier (Lines 49 to 52). We record the SLO cost on the tenant's SLA and record the metrics (Lines 53 and 54). When every page access request is processed, we total up the SLA violation penalties for every tenant, aggregate the metrics, and return the result (Lines 56 to 60).

IV. Experimental Evaluation

Now we demonstrate the CPBPSim framework (implemented with Python[4]) and its utility for the rapid development and testing of BP manager components.

TABLE I. Storage Tier Parameters

tier	*space (pages)*	*CPU access*	*SLO*	*access type*	*cost (ns)*	*access type*	*cost (ns)*
SSD	10^8	no	latency	read	25000	update	300000
NVM	$2.5 * 10^6$	yes	latency	read	100	update	100
RAM	$2.5 * 10^5$	yes	latency	read	50	update	50

A. Experiment Setup

We configure the simulator to use three storage tiers: SSD, NVM, and RAM (Table I). We allocate $2.5*10^5$ pages of free RAM space (2 GB, considering an 8 kB disk page), $2.5*10^6$ pages of free NVM space (20 GB), and 10^8 pages of SSD space

[4] https://github.com/BasiukTV/cpbpsim

Algorithm 6 CPBPSim

Input:
N - total number of page access requests; $t_{1..N}$ - page access request timestamps; $p_{1..N}$ - page access request page IDs; $ten_{1..N}$ - page access request tenant IDs; $type_{1..N}$ - page access request types; S - total number of BP storage tiers; $prms_{1..S}$ - storage tier parameters; $DAP_{1..S}$ - data admission policies; $DEP_{1..S}$ - data eviction policies; T - total number of tenants; $SLA_{1..T}$ - SLA violation penalty functions; $DMP_{1..T}$ - data migration policies; P - total number of data pages; $meta_{1..P}$ - data page metadata; *mon* - metrics monitor
Output:
$pen_{1..T}$ - total SLA violation penalties accrued

```
1.  function sim(t_1..N, p_1..N, ten_1..N, type_1..N)
2.    for i := 1 to N do
3.      tr := meta[p[i]].tier # Current page storage tier
4.      chain := [] # Initialize data access chain
5.      if !DAP[tr].should_admit(p[i]) and prms[tr].cpu_acc then
6.        chain.prepend(p[i], tr, type[i])
7.      else
8.        prms[tr].free_space++ # Page is about to free up space
9.        DEP[tr].update_residency(p[i], False)
10.       ntr := tr # New tier
11.       do # Admit to a new tier which has access to CPU cache
12.         ntr := DMP[ten[i]].dest_on_admit (t[i], p[i], ntr)
13.       while !prms[ntr].cpu_acc
14.       # Now make sure the destination tier has free space
15.       vp := p[i] # Victim page
16.       mtype := type[i] # Migration type
17.       while True
18.         chain.prepend(vp, ntr, mtype)
19.         if prms[ntr].free_space > 0 then
20.           prms[ntr].free_space--
21.           break
22.         else
23.           vp := DEP[ntr].evict()
24.           if meta[vp].dirty then
25.             ntr := DMP[ten[i]].dest_on_evict (t[i], vp, ntr)
26.             mtype := "copy"
27.           else
28.             break
29.           end if
30.         end if
31.       end while
32.     end if
33.     slo_val := 0
34.     for j := 1 to chain.len
35.       dp, dtr, dtp := chain[i] # Destination page, tier, and type
36.       slo_val := slo_val + prms[dtr][dtp][SLA[ten[i]].slo]
37.       if dtp == "copy" then
38.         sp, str, stp := chain[i+1] # Source page, tier, and type
39.         slo_val := slo_val + prms[str]["read"][SLA[ten[i]].slo]
40.       end if
41.       meta[dp].tier = dtr
42.       if dtr == "SSD" and meta[dp].dirty then
43.         meta[dp].dirty = False
44.       end if
45.       if dtr != "SSD" and dtp == "update" then
46.         met[dp].dirty = True
47.       end if
48.     end for
49.     for j := 1 to S do
50.       DAP[j].record_access(t[i], p[i], type[i])
51.       DEP[j].record_access (t[i], p[i], type[i])
52.     end for
53.     SLA[ten[i]].rec_slo(t[i], slo_val)
54.     mon.record_metric(t[i], ten[i], tr, chain, slo_val)
55.   end for
56.   for i := 1 to T do
57.     pen[i] := SLA[i].eval_penalty(0)
58.   end for
59.   mon.aggregate_metrics(0)
60.   return penalty_1..T
61. end function
```

(800 GB) to be used by the BP. We configure the latency SLO to be used by the simulator. The values for the "read" and "update" access type SLO costs are from [3].

We generated a single page access sequence to be used throughout the experiment for three different tenant data access types, which we called bronze, silver, and gold (Table II). The generated sequence represents eight hours of page accesses by three tenants of each type (over 10^7 total page accesses).

TABLE II. TENANT DATA ACCESS PATTERNS

tenant	*rate (requests/s)*	*pageID distribution*	*data size (pages)*	*read %*
bronze	5	uniformal	10^6	80
silver	25	normal(σ=$2*10^5$)	$2*10^6$	80
gold	100	Pareto(α=2)	10^7	80

For the tenant SLAs we used a simple step-based function with a single step (Table III). The function is evaluated every ten seconds, by averaging the latency SLO cost incurred during the period and comparing it with the "no-penalty cutoff" value. If the average latency is above the cutoff, the penalty is assigned.

TABLE III. TENANT SLAS

tenant	*SLO*	*evaluation period (s)*	*cost aggregation*	*no-penalty cutoff (ns)*	*penalty ($)*
bronze	latency	10	average	30000	0.01
silver	latency	10	average	30000	0.05
gold	latency	10	average	30000	0.2

B. Establishing the Baseline

To establish the baseline SLA violation penalty, we configure the simulator to use the FIFO-based data eviction policies for the RAM and NVM storage tiers, "always admit" data admission policies for the SSD and NVM tiers, and a data migration policy which has 50/50 chance of migrating data to NVM vs. RAM upon admission from SSD and 50/50 chance of migrating data to NVM vs. SSD upon eviction from RAM (data is always promoted from NVM to RAM and evicted from NVM to SSD). We evaluate the SLA after one hour of BP warmup time. The results are shown in the column "baseline" in Table IV.

TABLE IV. SIMULATION RESULTS

	baseline	*lru1*	*lru2*	*lru2-2q*	*lru2-2q-h*
average bronze penalty ($)	12.4	12.45	11.7	11.22	17.36
average silver penalty ($)	68.57	69.07	62.38	61.73	93.43
average gold penalty ($)	291.3	290.53	268.2	270.6	129.8
average total penalty ($)	1116.85	1116.15	1024.83	1030.66	721.77
runtime (s)	325	382	388	421	505

C. Use of Heterogeneous Data Eviction Policies

In order to demonstrate the use heterogeneous data eviction policies, we re-run the simulations after first reconfiguring the RAM storage tier to use the LRU-based data eviction policy (column "lru1" in Table IV), and then reconfiguring both the RAM and NVM tiers to use the same policy (column "lru2" in Table IV). The results indicate that the lru1 does not show much change as less than 10% of fast memory is affected. However, lru2 demonstrates a more significant reduction in the SLA violation penalty.

D. Use of Heterogeneous Data Admission Policies

To demonstrate the use of the heterogeneous data admission policies, we re-run the simulations after reconfiguring the NVM storage tier to use the 2Q-based [5] data admission policy (column "lru2-2q" in Table IV). Since 2Q is meant to slow down the admission of pages to RAM, the result of the simulation is a slight increase in the total penalty.

E. Use of Heterogeneous Data Migration Policies

Finally, to demonstrate the use of heterogeneous data migration policies, we configure bronze tenants to always promote data from SSD to NVM, and always evict data from RAM to SSD. We configure the gold tenants to always promote data from SSD to RAM, and always evict data from RAM to NVM. The simulation results are shown in the column "lru2-2q-h" in Table IV. As the gold tenants now prioritized in getting the RAM, their penalty is significantly reduced, while the penalty of the other tenants is slightly increased.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a framework for simulating the operation of the multi-tenant persistent memory-aware buffer pool manager, which provides a comprehensive overview of such BP manager components and their interfaces. We also proposed a simulator algorithm that orchestrates the interaction between the BP manager components. We then evaluated the said framework and algorithm to demonstrate their utility for future research. The proposed simulator has a shortcoming, which we plan to address in the future. It does not leverage the byte-addressability property of the NVM and RAM, which allows copying less than the entire pages of data between them.

REFERENCES

[1] S. Kuznetsov, "Towards a Native Architecture of In-NVM DBMS," in 2019 Actual Problems of Systems and Software Engineering.

[2] J. Liu and S. Chen, "Initial experience with 3D XPoint main memory," in 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW), 2019.

[3] J. Arulraj, A. Pavlo and K. T. Malladi, "Multi-Tier Buffer Management and Storage System Design for Non-Volatile Memory", 2019.

[4] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada and M. Sato, "Managing non-volatile memory in database systems," in Proceedings of the 2018 International Conference on Management of Data, Houston, TX, USA, 2018.

[5] L. Lersch, I. Oukid, I. Schreter and W. Lehner, "Rethinking DRAM caching for LSMs in an NVRAM environment," in European Conference on Advances in Databases and Information Systems, 2017.

[6] L. Lersch, W. Lehner and I. Oukid, "Persistent Buffer Management with Optimistic Consistency," arXiv preprint arXiv:1905.06760, 2019.

[7] F. J. Corbato, "A paging experiment with the multics system," MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[8] V. Narasayya, S. Das, M. Syamala, B. Chandramouli and S. Chaudhuri, "Sqlvm: Performance isolation in multi-tenant relational database-as-a-service," 2013.

[9] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala and S. Chaudhuri, "Sharing buffer pool memory in multi-tenant relational database-as-a-service," Proceedings of the VLDB Endowment, 2015.

[10] E. J. O'neil, P. E. O'neil and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," Acm Sigmod Record, 1993.