AutoTiering: Automatic Data Placement Manager in Multi-Tier All-Flash Datacenter

Zhengyu Yang†, Morteza Hoseinzadeh‡, Allen Andrews†, Clay Mayers‡, David (Thomas) Evans†, Rory (Thomas) Bolt†, Janki Bhimani*, Ningfang Mi† and Steven Swanson‡

† Dept. of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115
‡ Dept. of Computer Science and Engineering, University of California San Diego, San Diego, CA 92093
© Samsung Semiconductor Inc., Memory Solution Research Lab, Software Group, San Diego, CA 92121

Abstract—In the year of 2017, the capital expenditure of Flash-based Solid State Drivers (SSDs) keeps declining and the storage capacity of SSDs keeps increasing. As a result, the “selling point” of traditional spinning hard disk drives (HDDs) as a backend storage – low cost and large capacity – is no longer unique, and eventually they will be replaced by low-end SSDs which have large capacity but perform orders of magnitude better than HDDs. Thus, it is widely believed that all-flash multi-tier storage systems will be adopted in the enterprise datacenters in the near future. However, existing caching or tiering solutions for SSD-HDD hybrid storage systems are not suitable for all-flash storage systems. This is because all-flash storage systems do not have a large speed difference (e.g., 10x) among each tier. Instead, different specialties (such as high performance, high capacity, etc.) of each tier should be taken into consideration. Motivated by this, we develop an automatic data placement manager called “AutoTiering” to handle VM (virtual machine) allocation and migration in an all-flash multi-tier datacenter to best utilize the storage resource, optimize the performance, and reduce the migration overhead. AutoTiering is based on an optimization framework, whose core technique is to predict VM’s performance change on different tiers with different specialties without conducting real migration. As far as we know, AutoTiering is the first optimization solution designed for all-flash multi-tier datacenters. We implement AutoTiering on VMware ESXi [1], and experimental results show that it can significantly improve the I/O performance compared to existing solutions.

Index Terms—All-Flash Datacenter, Tiering Algorithm, Storage I/O, Cache, NVMe SSD, Big Data, Cloud Computing, Resource Management, Workload Evaluation & Estimation

I. INTRODUCTION

A basic credendum of cloud computing can be summarized as: user devices are light terminals to assign jobs and gather results, while all heavy computations are conducted on remote distributed server clusters. This light-terminal-heavy-server structure makes high availability no longer an option, but a requirement in today’s datacenters. Furthermore, when we bring compute, network, and storage capabilities into balance, it is found that the biggest challenge here is closing the gap between compute and storage performance to shift storage’s curve back towards Moore’s law [2], [3]. In other words, storage I/O is the biggest bottleneck in large scale datacenters. As shown in studies [4] and [5], the time consumed to wait for I/Os is the main cause of idling and wasting CPU resources, since lots of popular cloud applications are I/O intensive, such as video streaming, file sync and backup, data iteration for machine learning, etc.

To solve the problem caused by I/O bottlenecks, parallel I/O to multiple HDDs in Redundant Array of Independent Disks (RAID) becomes a common approach. However, the performance improvement from RAID is still limited, therefore, lots of big data applications strive to store intermediate data to memory as much as possible such as Apache Spark. Unfortunately, memory is too expensive, and its capacity is very limited (e.g., 64~128GB per server), so it alone is not able to support super-scale cloud computing use cases. As a device between RAM and HDD, SSD has an acceptable price and space. Since 2008, SSDs started to be adopted in the server market and broke the asymmetric R/W IOPS barrier dramatically, and became one of the promising solutions to speedup storage systems as a cache or a fast tier for slow HDDs. However, as time goes by, SSD-HDD based solutions are no longer competent to meet the current big data requirements, due to the huge I/O speed gap between SSDs and HDDs. On the other hand, the capital expenditure of Flash-based SSDs keeps decreasing and the storage capacity of SSDs keeps increasing. Thus, it is widely believed that SSD-HDD solution is just for a transition period, and all-flash multi-tier storage systems will be adopted in the enterprise datacenter in the near future, similar to what happened to HDD-tape solution 30 years ago. For example, high end NVMe SSDs can replace SATA SSD, and low end TLC SSDs will replace HDDs.

However, traditional caching algorithms are deemed useful only when the performance difference between storage devices is at least 10x, while the gap between SSD tiers in all-flash datacenters is not that big. More importantly, SSDs are expensive, and it is costly to maintain duplicated copies (one for cache and one for backend data) in two SSD tiers. Thus, we develop an automatic data placement manager called “AutoTiering” to handle VM allocation and migration in an all-flash multi-tier datacenter. The ultimate goal is to best utilize the storage resource, obtain optimize the performance, and reduce the migration overhead, by associating VMs with an appropriate tier of storage. AutoTiering is based on an optimization framework to provide the best global (i.e., for all VMs in the datacenter) migration and allocation solution over runtime. AutoTiering’s approximation approach further solves the simplified problem in a polynomial time, which considers both historical and predicted performance factors, as well as estimated migration cost. This comprehensive methodology prevents to frequently migrate back and forth VMs between
tiers due to I/O spikes. Specifically, AutoTiering uses a micro-
benchmark-based sensitivity calibration and regression session
to predict VM’s performance change on different tiers without
performing real migration, since different VMs may have
different benefits of being upgraded to a high end tier. We
implement AutoTiering on VMware ESXi [1] and evaluate
its performance with a set of representative applications. The
experimental results show that AutoTiering can significantly
improve the I/O performance by increasing I/O throughput
and bandwidth as well as reducing I/O latency.

The rest of this paper is organized as follows. Sec. II
presents the background and literature review. Sec. III formu-
lates the problem and introduces an optimization frame-
work. Sec. IV proposes our approximation algorithm to solve
the problem in a polynomial time. Experimental evaluation
results and analysis are discussed in Sec. V. We present the
conclusions in Sec. VI.

II. BACKGROUND AND LITERATURE REVIEW
Substantial work has been done to improve I/O operations
in datacenters at both hardware and software levels. In this
section, we discuss some of these work as well as the
evolutionary inclination towards AutoTiering.

SSD as Cache/Tier of HDD: In the recent years, Flash-
based SSDs have been commonly used in datacenters. An
SSD can be used whether as a cache for HDD or as a distinct
storage tier. The main difference is that the cache approach has
two copies for hot data, one in SSD and one in HDD (two
copies are synced under the write through policy, and are not
synced under the write back policy), while the tiering approach
simply migrates data between tiers and only keeps one version
of the dataset. There are lots of caching algorithms [6]–[9]
and tiering mechanisms [5], [8], [10], [11] are used in cloud
storage systems.

Data Placement in Multi-tier All Flash Data Center: SSD-
HDD based solutions may work for a limited number of
users (VMs) with mediate I/O intensity and small working
set size, but for the era of super-scale clusters (e.g., clouding
computing, IoT in 5G network), the I/O bottleneck gets
mitigated, but not resolved [12]. The main reason is that in
both SSD-HDD caching and tiering approaches, there still
exists a huge performance gap between SSDs and HDDs.
With the decreasing price and increasing capacity of SSDs,
a promising solution to quench this gap is to setup an all-
flash datacenter which is becoming a reasonable solution in
the near future. Our previous work [13] developed a workload
dispatcher for all-flash datacenters to minimize the total cost
of ownership. With the aim of further reducing the overall
capacity, study [14] proposed to periodically recompute VM
assignments in every few hours. Furthermore, [15] introduced
a mathematical model formulation for migrating VMs among
tiers, with the aim of minimizing the overhead of data mi-
gration. We summarize the specs of SSDs with different ends
available in market by July 2017 in Table I. As we can see, an
all-flash multi-tiers solution can be built from different SSDs
with different specialties, e.g., super performance tier with
3D XPoint SSD, high performance tier with NVMe and SLC
SSDs, and large capacity tier with MLC and TLC SSDs.

III. PROBLEM FORMULATION
In this section, we first formulate the problem of VM
allocation and migration in an all-flash multi-tier datacenter,
and then develop an optimization framework to best utilize
SSD resource among VMs in the datacenter.

A. System Architecture

Fig. 1 illustrates the system architecture of AutoTiering
which has the following components:

- **IO Filter:** Attached to each VMDK being managed on
each host, it is responsible for collecting I/O related
statistics as well as running special latency tests on
every VMDK. The data will be collected and sent to the
AutoTiering Daemon on the host.

- **Daemon:** Running on the VM hypervisor of all hosts,
it tracks the workload change (i.e., I/O access pattern
change) of each VM, collects the results of latency
injection tests from the IO Filter, and sends them to the
Controller.

- **Controller:** Running on a dedicated server, it is respon-
sible for making decisions to trigger migration based on
the predicted VM performance (if is migrated to other
tiers) and the corresponding migration overhead.

B. Optimization Framework

To develop an optimization framework aimed at minimizing
the total amount of server hours by determining a VM mi-
gration schedule, we formulate the problem by investigating
the following factors: First, from each VM’s point of view,
the reason for a certain VM to be migrated from one tier
to another is that the VM can perform better (e.g., less
average I/O latency, higher IOPS, higher throughput, etc.)
after migration. Second, the corresponding migration cost need
to be considered, because migration is relatively expensive
(consumes resource and time) and not ignitable. Third, from the
global optimization’s point of view, it is hard to satisfy
all VMs to be migrated to their favorite tiers at the same
time due to resource constraints and their corresponding SLAs (i.e.,
Service Level Agreement). Fourth, the global optimization
should consider overtime changes of all VMs as well as post-
effects of migration. For example, the current best allocation
solution may lead to a bad situation for the future since VMs
are changing behaviors during runtime. Based on these factors,
our optimization framework needs to consider potential bene-
fits and penalties, migration overhead, historical and predicted

---

<table>
<thead>
<tr>
<th>SSD Type</th>
<th>Cost ($/GB)</th>
<th>Max Size (GBytes)</th>
<th>Read Time (µs)</th>
<th>Write Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D XPoint</td>
<td>4.50</td>
<td>3.25G</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>NVMe</td>
<td>0.57</td>
<td>3.2T</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>SLC SATA3</td>
<td>0.64</td>
<td>4.80G</td>
<td>25</td>
<td>250</td>
</tr>
<tr>
<td>TLC SATA3</td>
<td>0.30</td>
<td>2T</td>
<td>50</td>
<td>750</td>
</tr>
<tr>
<td>3D XPoint</td>
<td>0.28</td>
<td>3.84T</td>
<td>75</td>
<td>1125</td>
</tr>
</tbody>
</table>
performance of VMs on each tier, SLA, as shown in Eq. 1 to 5. Table II represents notations that we use in this paper.

Maximize:

$$\sum_{\forall v, \forall \tau} w_{v, \tau} \cdot \left[ \sum_{\forall k} \alpha_k \cdot r(v, t_{v, \tau}, \tau, k) - \beta \cdot g(v, t_{v, \tau - 1}, t_v, \tau) \right],$$

Subject to:

for \( \forall v, \forall \tau : \)

\( t_{v, \tau} \neq \emptyset, \) and \( |t_{v, \tau}| \geq 1, \)

for \( \forall t_1, t_2 \in [0, +\infty) : \)

\( r(v, t_{v, t_1}, t_{v, t_2}, k) = r(v, t_{v, t_1}, t_{v, t_2}, t_1, k) \geq 0, \)

for \( \forall v, \forall \tau : \)

\( r(v, t_{v, \tau}, \tau, k) = r_{Prd}(v, t_{v, \tau - 1}, t_{v, \tau}, \tau - 1, k) \geq 0, \)

\( \sum_{\forall v} r(v, t_{v, \tau}, \tau, k) \leq \Gamma_k \cdot R(t_{v, \tau}, k), \)

The main idea is to maximize the “Profit”, which is the entire performance gain minus penalty, i.e., “Performance Gain - Performance Penalty”, as shown in the objective function Eq. 1. The inner “sum” operator conducts a weighted sum of the usage of all types of resources (such as throughput, bandwidth, and storage size, etc.) of each VM, assuming migrating VM \( v \) from tier \( t_{v, \tau - 1} \) to \( t_{v, \tau} \). The outer “sum” operator further iterates all possible migration cases, where weight parameter \( w_{v, \tau} \) reflects the SLA of VM \( v \) in \( \tau \) epoch. Notice that the term “migration” in this paper is not to migrate a VM from one host server to another. Instead, only backend VMDK files are migrated from one to the other SSD tier. As a result, non-disk I/O related resources (e.g., host-side CPU and memory) are not considered in this paper. Eq. 2 guarantees that each VM is hosted by at least one disk tier. In fact, each VM can have multiple VMDKs, and each of them can be located at different tiers. Unlike the previous SSD-HDD tiering work [5], [11] that are operating on fine-grained data blocks due to HDD speed bottleneck, the minimal unit to migrate in this paper is the entire VMDK of each VM. Eq. 3 ensures that storage size (i.e., VMDK size) will not change before and after migrations, where \( k_v \) is the type index of storage resource. Eq. 4 shows that a prediction model function is utilized to predict the performance gain (for details, see Sec. IV-2). Eq. 5 lists resource constraints, where \( \Gamma_k \) is preset upper bound (in percentage) of each type (i.e., \( k \)-th) of resource that can be used. Finally, the temporal migration overhead is the size of the VM to be migrated, divided by the bottleneck of migrate-out read speed, and migrate-in write speed, i.e., \( g(v, t_{v, \tau - 1}, t_{v, \tau}) = \frac{r(v, t_{v, \tau - 1}, \tau - 1, k_v)}{\mu(v, \tau - 1) - \beta}. \) The migration speed is \( \mu(v, \tau - 1) = min[(Pr(A, t_{v, \tau - 1}, \tau - 1) + Pw(A, t_{v, \tau - 1}, \tau - 1)], Pr(A, t_{v, \tau - 1}, \tau - 1) \) is the function of available remaining read throughput. Since we are going to live migrate VM \( v \), the read throughput used by this VM \( (Pr(t_{v, \tau - 1}, \tau - 1)) \) is also available and has been added back here. \( Pw(A, t_{v, \tau - 1}, \tau - 1) \) gives the migrate-in write throughput.

Since the system has no information on the future workload I/O patterns, it is impossible to conduct the global optimization for all \( \tau \) time periods during runtime. Moreover, in the global optimal solution, the decision making process for each migration epoch theoretically depends on the past and future epochs. It means that Eq. 1 cannot be solved by traditional sub-optimal-based dynamic programming techniques. Lastly,

**TABLE II: Notations.**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v, v_i )</td>
<td>VM ( v ), ( i )-th VM, ( v \in [1, v_{max}] ), where ( v_{max} ) is the last VM.</td>
</tr>
<tr>
<td>( t, t_i )</td>
<td>Tier ( t ), ( i )-th tier, ( t \in [1, t_{max}] ), where ( t_{max} ) is the last tier.</td>
</tr>
<tr>
<td>( t_{v, \tau} )</td>
<td>VM ( v )’s hosting tier during epoch ( \tau ).</td>
</tr>
<tr>
<td>( k )</td>
<td>Different types of resources, ( t \in [1, k_{max}] ), such as throughput, bandwidth, storage size, and etc.</td>
</tr>
<tr>
<td>( \tau )</td>
<td>Temporal epoch ID, where ( \tau \in [0, +\infty) ).</td>
</tr>
<tr>
<td>( r(v, t_{v, \tau}, \tau, k) )</td>
<td>Predicted type of ( k ) resource usage of VM ( v ) running on tier ( t_{v, \tau} ).</td>
</tr>
<tr>
<td>( g(v, t_{v, \tau - 1}, t_{v, \tau}) )</td>
<td>Migration cost of VM ( v ) during epoch ( \tau ) from tier ( t_{v, \tau - 1} ) to tier ( t_{v, \tau} ).</td>
</tr>
<tr>
<td>( k_v )</td>
<td>“Storage” resource’s index.</td>
</tr>
<tr>
<td>( \mu(v, \tau - 1) )</td>
<td>Estimated migration speed of VM ( v ) at time ( \tau - 1 ) epoch.</td>
</tr>
<tr>
<td>( P_{Prd}(v, t, \tau) ), ( P_{bw}(v, t, \tau) ), ( P_{bw}(v, t, \tau) )</td>
<td>All remaining available read/write resource of tier ( t ) during epoch ( \tau ).</td>
</tr>
<tr>
<td>( P_{Prd}(A, t, \tau) ), ( P_{bw}(A, t, \tau) )</td>
<td>All remaining available read/write resource of tier ( t ) during epoch ( \tau ).</td>
</tr>
<tr>
<td>( \Gamma_k )</td>
<td>Upper bound (in percentage) of each type of resource that can be used.</td>
</tr>
<tr>
<td>( R(t_{v, \tau}, k) )</td>
<td>Total capacity of ( k )-th type of resource on tier ( t_{v, \tau} ).</td>
</tr>
<tr>
<td>( L_t )</td>
<td>Original average I/O latency (without injected latency) of tier ( t ).</td>
</tr>
<tr>
<td>( b_v, m_v )</td>
<td>Parameters of TSSCS liner regression model ( (y = mx + b) ).</td>
</tr>
<tr>
<td>( s_v )</td>
<td>Average I/O size of VM ( v ).</td>
</tr>
<tr>
<td>( S_v, v \cdot m_{max} )</td>
<td>Storage size of VM ( v ).</td>
</tr>
<tr>
<td>( w_{v, \tau}, \frac{wetP[t]}{wetB[t]}, \frac{wetS[t]}{wetS[t]} )</td>
<td>Weight of VM ( v ) and weights of tier ( t )’s each type of resource.</td>
</tr>
<tr>
<td>( \frac{maxP[t]}{maxB[t]}, \frac{maxS[t]}{maxS[t]}, \frac{spec[t, P]}{spec[t, B, spec[t]} )</td>
<td>Preset available resource caps and specialties of tier ( t ), ( P )=throughput, ( B )=bandwidth, ( S )=storage size.</td>
</tr>
</tbody>
</table>
IV. APPROXIMATION ALGORITHM DESIGN

To obtain the result close to the optimized solution in a polynomial time, we have to relax some constraints. In detail, we first downgrade the goal from “global optimizing for all time” to “only optimizing for each epoch” (i.e., runtime greedy). Furthermore, since we have the foreknowledge of each tier’s performance “specialty” (such as high throughput, high bandwidth, large space, small write amplification function, large over-provisioning ratio, large program/erase cycles, etc.), we can make migration decisions based on the ranking of the estimated performance of each VM on each tier, considering each tier’s specialties and corresponding estimation of migration overhead. Details of our approximation algorithm are discussed in the following subsections.

1) Main Procedure: Alg. 1 lines 1-8 show the main procedure of AutoTiering, which periodically monitors the performance and checks whether a VM needs to be migrated. Specifically, monitorEpoch is the frequency of evaluating andregressing the performance estimation model, and migrationEpoch is the frequency of triggering VM migration from one tier to the other one. The migration decision is made at the beginning of each migrationEpoch, which is greater than monitorEpoch. Apparently, the smaller temporal window sizes, the more frequent monitoring, measurement, and migration. The system administrator can balance a tradeoff between the accuracy and the migration cost by conducting a sensitivity analysis before deployment. As shown in line 4 of Alg. 1, procedure tierSpdSenCalibrate estimates VM’s performance on each tier based on the regression model. Line 5, thus, calculates performance matrices, and line 6 calculates the score by considering the historical and current performance of each VM, estimates the corresponding migration overhead. Finally, VM migrations are triggered, see line 8. We describe their details in the following subsections.

2) Tier Speed Sensitivity Calibration: In order to estimate VM’s performance on other tiers without conducting actual migration, we first try to “emulate” the speed of tiers by manually injecting a synthetic latency to each VM’s I/Os, and measure the resultant effect on total I/O latency by calling IOFilter APIs. Our preliminary experiments show that the performance variation can be modeled to a linear function.

IOFilter APIs. Our preliminary experiments show that the performance variation can be modeled to a linear function.

Algorithm 1: AutoTiering Procedure Part I.

```
1 Procedure autoTiering()
2     while True do
3         if currTime MOD monitorEpoch = 0 then
4             tierSpdSenCalibration();
5             calCapacityMatrices();
6             calScore();
7         end
8         if currTime MOD migrationEpoch = 0 then
9             triggerMigration();
10        end
11     end

9 Procedure tierSpdSenCalibration()
10     for t ∈ tierSet do
11         for v ∈ VMSet[t] do
12             for samplesWithLatency ∈ sampleSet[t][v] do
13                CV = calCV(samplesWithLatency); samplesWithLatency = avg(samplesWithLatency); CV = len(sampleSet[t][v]);
14                if CV ≥ 1 then
15                    VM[v].conf = 0.05;
16                else
17                    VM[v].conf = 1 - CV;
18                    (VM[v].m, VM[v].b) = regress(sampleSet[t][v]);
19                end
20        end
21     end

14 Procedure calCapacityMatrices()
15     for t ∈ tierSet do
16         for v ∈ VMSet[t] do
17             for v,t do
18                 Lat = estimateAvgLat(v,t);
19                 if Lat > 0 then
20                     IOPS = 10^6/Lat;
21                 else
22                     IOPS = 0;
23                 VMCapMat[t][v].P = IOPS;
24                 VMCapMat[t][v].B = (IOPS × VM[v].avgIOSize);
25                 VMCapMat[t][v].S = VM[v].size;
26             end
27     end

17 Procedure estimateAvgLat(v,t)
18     return VM[v].m × (tierLatency[v] − tierLatency[VM[v].[tier]]) + VM[v].b;
```

frequency, without affecting the current hosting workload performance.

- **Multi-Samples per Latency**: TSSCS improves the accuracy of emulating each VM’s performance under each tier by taking the average over multiple samples that are obtained with the same injected latency of each tier.
- **Multi-Latencies per Session**: TSSCS takes multiple latencies per session to refine the regression.
- **Multi-Sessions during Runtime**: TSSCS is periodically triggered to update the regression model by calling “tierSpdSenCalibration” in Alg. 1 line 4.

Fig. 2 depicts an example of three VMs running on three different tiers: VM v1 on tier t1, VM v2 on tier t2, and VM v3 on tier t3. Assume t1 is 2,000 µs faster than t2, and t2 is 2,000 µs faster than t3. We run TSSCS on each VM on their hosting tier, and get the latency curves shown in three plots in Fig. 2. Since all injected latencies are additional to the original bare latency (i.e., without injected latency), we have to align them according to the absolute latency values (i.e., bare latency + injected latency). Notice
that since we cannot inject negative latencies (i.e., obviously we can only slow down the tier), the dash lines in subfigures “VM2 on tier2” and “VM3 on tier3” are regressed based on the solid lines. After that, we can draw three (colored) lines for each tier based on their absolute latency values (i.e., red for tier1, green for tier2, and blue for tier3). Then, we can easily predict the average I/O latency values of each VM on each tier (i.e., red points for t1, green points for t2, and blue points for t3). We see that VM1 is the most sensitive one to tier speed changes (i.e., with the greatest gradient), while VM3 is the least sensitive one (i.e., relatively flat). Therefore, intuitively, we should assign VM1 to tier1 (fastest tier) and VM3 to tier3 (slowest tier). Alg. 1 lines 9-21 describe the procedure of tier speed sensitivity calibration session (TSSCS). In addition, AutoTiering calculates the coefficient variation (CV) of sampling results to decide the estimation confidence, see in lines 13 and 16-19.

3) Performance Capacity Matrices: Once we have the average latency vs. injected latency curves of each VM of the current moment, we calculate corresponding performance estimation of throughput (denoted as P, unit in IOPS), bandwidth (denoted as B, unit in MBPS), and storage size (denoted as S, unit in bytes), and record them into three two-dimensional matrices, i.e., $VMCapMat[t][v].P$, $VMCapMat[t][v].B$, and $VMCapMat[t][v].S$, where t and v are IDs of tier and VM, respectively. Compared to the first two matrices, the last matrix is relatively straightforward to be obtained by calling the hypervisor APIs to measure the storage size that each VM is occupying. As shown in Alg. 1 lines 22-33, AutoTiering updates the VM capacity matrices (i.e., $VMCapMat$) by reiterating for all tiers and VMs. It estimates “new” latency under other tiers by calling the “estimateAvgLat(v, t)” function in Alg. 1 line 25. Lines 34-35 show the detail of $estimateAvgLat$ function, where the input parameters are VM v and target tier t for estimation, which returns an estimation based on linear regressions of m and b values. Once the estimated average I/O latency results are obtained, we calculate the throughput and bandwidth in Alg. 1 lines 30 and 31. Lastly, the storage size will also be updated into the $VMCapMat$ (i.e., Alg. 1 line 32). Furthermore, since it gets harder to evaluate demands of different recourse types together (because they have different units), AutoTiering normalizes the VM’s estimated/measured throughput, bandwidth and storage utilization value according to the total available resource capacity of each tier, which is called the normalized capacity utilization rate matrix $VMCapRateMat$, as shown in Alg. 2 lines 4-13.

4) Performance Score Calculation: AutoTiering takes three steps to calculate the performance score based to reflect the following factors: First, Characteristics of both tier and VM: the score should reflect each tier’s specialty and each VM’s workload characteristics running on each tier. Thus, our solution is to calculate each VM’s score on each tier separately. Second, SLA weights: VMs are not equal since they have different SLA weights, as shown in Eq. 1. Third, Confidence of estimation: we use coefficient variation calculated in performance matrices to reflect the confidence of estimation. Fourth,
History and migration costs: a convolutional aging factor for historical scores and estimated migration cost are considered during the score calculation.

[Step 1] Tier Specialty Matrix: To reflect the specialty, we introduce a two-dimension tier-satellite matrix \( spc \). For example, \( spc[t],P = 1, spc[t],B = 1 \) and \( spc[t],S = 0 \) reflects that tier \( t \) is good at throughput and bandwidth, but bad at storage capacity. In fact, this matrix can be extended to a finer granularity to further control specialty degree, and more types of resources can be included into this matrix, if needed. Moreover, tiers are sorted in the order of high-to-low-end (e.g., most-to-least-expensive-tier) in the matrix, and this order is regarded as a priority order during the migration decision making period.

[Step 2] Orthogonal Match between VM Demands and Tier Specialties: The next question is “how to reflect each VM’s performance on each tier AND reflect how good VMs can utilize each tier’s specialty?”. Our solution is to introduce a process called “orthogonal match” (denoted as “\( \Omega \)”) to score the “matchness”. This process is a per-VM-per-tier multiplication operation of “specialty” matrix and “\( VM\text{CapRateMat} \)” matrix, i.e.,

\[
currCapScore(t, v) = \Omega(t, v) = \left[ spc[t],P \times wetP[t], spc[t],B \times wetB[t], spc[t],S \times wetS[t] \right] \times \left[ VM\text{CapRateMat}[t][v],P \cdot \text{VM}[v],\text{SLA} \times \text{VM}[v],\text{conf} \right] \times \left[ VM\text{CapRateMat}[t][v],B \times \text{VM}[v],\text{perf} \right] \times \left[ VM\text{CapRateMat}[t][v],S \right] \\
\times \left( wetP[t] + wetB[t] + wetS[t] \right)
\]

where \( currCapScore \) gives the current capacity score, and \( VM\text{CapRateMat} \) is the VM capacity utilization rate matrix.

[Step 3] Convolutional Performance Score: The final performance score is a convolutional sum of historical score, current epoch capacity score and penalty of corresponding migration cost, i.e.,

\[
tierVMPerfScore = agingFactor \times \text{histTierVMPerfScore} + currCapScore – \text{wetMig} \times \text{migCost}
\]

This process is also shown in Alg. 2 line 14. Specifically, to avoid the case that some VMs are frequently migrated back and forth between tiers (due to making decision only based on recent one epoch which may contain I/O spikes or bursts), AutoTiering needs to convolutionally consider history scores, with a preset \( agingFactor \) to fadeout outdated scores. Current capacity score \( currCapScore \) is calculated by the orthogonal match procedure. Additionally, Alg. 2 lines 16-18 show the procedure of \( migrationCost \) calculation.

### V. Performance Evaluation

#### A. Evaluation Methodology

1) Implementation Details: We build AutoTiering on VMware ESXi hypervisor 6.0.0 [1]. Table III summarizes the server configuration of our implementation. Table IV further shows the specs of each tier (each tier has multiple SSDs). We set the specialty matrix such that tier 1 is good for throughput and bandwidth performance, tier 2 is the secondary performance tier but with larger capacity, and tier 3 is the capacity tier to replace HDDs.

2) Workloads: To evaluate performance under different algorithms, we use IOMeter [16] and FIO [17] to generate I/O workloads to represent real world use cases. Table V shows some statistical analysis of 14 used workloads. Each VM has multiple VMDKs with different sizes, such as system disk, datastore disk, and etc.

### Table III: Host server configuration.

<table>
<thead>
<tr>
<th>Component</th>
<th>Specs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Server</td>
<td>HPE ProLiant DL380 G9</td>
</tr>
<tr>
<td>Host Processor</td>
<td>Intel Xeon CPU E5-2680 v3</td>
</tr>
<tr>
<td>Host Processor Speed</td>
<td>2.40GHz</td>
</tr>
<tr>
<td>Host Processor Cores</td>
<td>12 Cores</td>
</tr>
<tr>
<td>Host Memory Capacity</td>
<td>64GB DIMM DDR4</td>
</tr>
<tr>
<td>Host Memory Data Rate</td>
<td>2133 MHz</td>
</tr>
<tr>
<td>Host Hypervisor</td>
<td>VMware ESXi 6.0</td>
</tr>
</tbody>
</table>

### Table IV: Multi-tier flash drivers configuration.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Model</th>
<th>Protocol</th>
<th>IOPS</th>
<th>MBPS</th>
<th>PerDisk Size(GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Samsung PM953</td>
<td>NVMe</td>
<td>290K</td>
<td>19K</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>Samsung PM1633</td>
<td>SAS</td>
<td>200K</td>
<td>7K</td>
<td>1400</td>
</tr>
<tr>
<td>3</td>
<td>Samsung PM863</td>
<td>SATA</td>
<td>99K</td>
<td>18K</td>
<td>540</td>
</tr>
</tbody>
</table>

### Table V: Resource demands of selected workloads.

<table>
<thead>
<tr>
<th>Load</th>
<th>Workload</th>
<th>Represented Scenarios</th>
<th>Thrust. (IOPS)</th>
<th>BW. (BPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heavy</td>
<td>BasicVerify</td>
<td>SQL database server</td>
<td>95.5K</td>
<td>373M</td>
</tr>
<tr>
<td></td>
<td>SSDSteady</td>
<td>System development</td>
<td>116K</td>
<td>453M</td>
</tr>
<tr>
<td></td>
<td>Zipt IOs</td>
<td>Web apps</td>
<td>1942K</td>
<td>7585M</td>
</tr>
<tr>
<td></td>
<td>AsyncRead</td>
<td>Read intensive apps</td>
<td>88.3K</td>
<td>345M</td>
</tr>
<tr>
<td></td>
<td>AsyncWrite</td>
<td>Write intensive apps</td>
<td>66.6K</td>
<td>258M</td>
</tr>
<tr>
<td>Middle</td>
<td>Flow</td>
<td>Big data frameworks</td>
<td>19.2K</td>
<td>120M</td>
</tr>
<tr>
<td></td>
<td>IOmeter</td>
<td>File server</td>
<td>47.4K</td>
<td>205M</td>
</tr>
<tr>
<td></td>
<td>JESD</td>
<td>High endurance apps</td>
<td>18.3K</td>
<td>136M</td>
</tr>
<tr>
<td></td>
<td>LatencyProfile</td>
<td>Cloud system manager</td>
<td>39.6K</td>
<td>155M</td>
</tr>
<tr>
<td></td>
<td>SSDTest</td>
<td>Hardware development</td>
<td>47.4K</td>
<td>205M</td>
</tr>
<tr>
<td>Light</td>
<td>RandZone</td>
<td>Multi-user database</td>
<td>7.75K</td>
<td>30.3M</td>
</tr>
<tr>
<td></td>
<td>SurfaceScan</td>
<td>Enterprise backup server</td>
<td>6.98K</td>
<td>436M</td>
</tr>
<tr>
<td></td>
<td>SyncRead</td>
<td>Read intensive sync apps</td>
<td>6.65K</td>
<td>25M</td>
</tr>
<tr>
<td></td>
<td>SyncWrite</td>
<td>Metadata sync server</td>
<td>4</td>
<td>10K</td>
</tr>
</tbody>
</table>

### B. Study on Throughput, Bandwidth and Latency Changes

Fig. 3 illustrates the average throughputs, bandwidth, and normalized latency of all tiers over time for both read (\( Rd \)) and write (\( Wt \)) I/Os. AutoTiering achieves up to 44.74% and 38.78% higher IOPS than IDT and EDT. Similar results can be obtained for bandwidth and latency as shown in Figs. 3(b) and (c). Fig. 4 depicts per-tier results to further show the performance improvement brought by AutoTiering. We observe that AutoTiering performs the best in terms of (both read and write) throughputs, bandwidths and latencies on tier 1, which is because the specialty matrix sets tier 1 to optimize
mention that AutoTiering achieves the lowest latencies in all cases except write latency in tier 2 (as shown in Fig. 4(c) 5th column), because AutoTiering migrates many VMDKs that have large average I/O size (high write bandwidth), and thus, as a tradeoff, the latency is increased. Moreover, Fig. 5 depicts the distribution of total throughput and bandwidth of all tiers for different algorithms. From Fig. 5(a), we observe that under AutoTiering (red curve), majority of I/Os has more than 100K IOPS and even half of them have more than 125K IOPS. In contrast, 90% of I/Os are less than 100K IOPS (blue curve) under IDT, and almost all I/Os from EDT are less than 100K (green curve). Similarly, from Fig. 5(b), we can see that the majority (around 90%) of IDT and EDT I/Os are less than 1,200 MBPS, while more than half of AutoTiering’s I/Os can achieve larger than 1,200 MBPS bandwidth.

C. Study on Runtime Distribution of Resource Utilization

Fig. 6 shows runtime changes of throughput, bandwidth and latency distribution across tiers over time. We observe that the areas in (c) and (f) are larger than those in (a)-(b), and (d)-(f), respectively. Area in (i) is also much smaller than those in (g)-(h). This verifies our observations in Sec. V-B that AutoTiering achieves better throughput and bandwidth performance than IDT and EDT. We also observe in (a) to (f) that area of each tier in AutoTiering is “thicker” than that in IDT and EDT (after AutoTiering’s warming up periods from 0-3 epochs). This indicates that AutoTiering can better utilize throughput and bandwidth resources of each tier by proper and less migrations. In (i), we see that the majority of the latency of AutoTiering is located in tier 3 (the write latency “T3 Rd Lat”), which is because that tier 3 is regarded as the capacity tier to replace HDD. As a result, AutoTiering migrates read-intensive VMs with large VMDKs to leverage tier 3, and leaves tiers 1 and 2 for other write-intensive workloads.

D. Study on Migration Overhead

To investigate the migration overhead of three algorithms, we show the normalized temporal migration cost results in Fig. 7. The blue bars show the normalized total migrated data size, and the green bars show the normalized number of VMs that are migrated (multiple migrations on a single VM is counted as 1). The former is to reflect the “working volume size” and the latter is to reflect the “working set size”. AutoTiering performs best among the three, since it migrates less data and interrupts less VMs, which saves lots of system resources. In summary, AutoTiering chooses the best tier for each VM for better performance and prevents unnecessary migrations due to I/O spikes, ascribed to its comprehensive performance-sensitive workloads. On the other hand, we also see that AutoTiering sometimes achieves lower throughput and bandwidth in tier 2 and 3 compared with IDT and EDT. This is because IDT is IOPS-only algorithm, which migrates high-IOPS-demand (especially write I/O) workloads to tier 1, such that the write IOPS is optimized. Similarly, EDT considers both IOPS and capacity, and thus has slightly better write IOPS compared to AutoTiering in the capacity tier 3. It is worth to
decision which is based on a more accurate performance estimation method.

VI. CONCLUSION

We present a novel data placement manager “AutoTiering” to optimize the virtual machine performance by allocating and migrating them across multiple SSD tiers in the all-flash data-center. AutoTiering is based on an optimization framework to provide the global best migration and allocation solution over runtime. We further proposed an approximation algorithm to solve the problem in a polynomial time, which considers both historical and predicted performance factors, and estimated migrating cost. Experimental results show that AutoTiering can significantly improve system performance. In the future, we plan to use kernel APIs to replace injected latency sampling process to further reduce performance estimation overhead.

REFERENCES