Transaction-aware SSD Cache Allocation for the Virtualization Environment

Zhen Tang*†, Heng Wu*†¶, Lei Sun‡, Zhongshan Ren*†, Wei Wang*†, Wei Zhou§ and Liang Yang§
*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
†University of Chinese Academy of Sciences ‡Tianjin Massive Data Processing Technology Laboratory §KSYUN
{tangzhen12,wuheng,renzhongshan13,wangwei}@otcaix.iscas.ac.cn, sunlei@bjsasc.com, {zhouwei2,yangliang1}@kingsoft.com
¶Corresponding author

***Abstract*—Flash-based Solid State Disk (SSD) is widely used in the Internet-based virtual computing environment, usually as cache of the hard disk drive-based virtual machine (VM) storage. Existing SSD caching schemes mainly treat the VMs as independent units and focus on critical performance metrics concerning one single VM, such as the IO latency, throughput, or the cache miss rate. However, in the Internet-based virtual computing environment, one transactional application usually consists of multiple VMs on different hypervisors. Transaction-aware SSD caching schemes may potentially better improve the end user-perceived quality of service. The key insight here is to utilize the relationships among VMs inside the transactional application to better guide the allocation of the SSD cache, so as to help learn the pattern of workload changes and build adaptive SSD caching schemes. To this end, we propose the *Transaction-Aware SSD caching (TA-SSD)*, which takes the characteristics of transactions into consideration, uses closed loop adaptation to react to changing workload, and introduces the genetic algorithm to enable nearly optimal planning. The evaluation shows that comparing to the equally partitioned cache, the allocation produced by the TA-SSD can boost the performance by up to 40%, with dynamic changes in the intensity and the type of the workload.**

***Index Terms*—SSD; Cache; Virtualization**

I. Introduction

Transactional applications are widely used in the Internet-based virtualization environment[1], with different VMs hosted on different cloud providers to balance the cost and performance[2]. The hypervisors, where the cloud provider hosts the VMs on, are mostly attached with a hard disk drive (HDD) based back end shared storage to store VM images. The IO performance is vital to the efficient use of virtualized resources.

Flash-based Solid State Disk (SSD) has recently been a widespread solution to improving the IO performance[3][4]. SSD has the significant advantage over HDD, especially for random IO operations. In the virtualization environment, SSD is often used as the cache for VMs on the hypervisor. The IO operations will first arrive at the SSD cache. Upon a cache hit, the operation will be quickly served by the cached data. Otherwise, it will be served by the much slower back end storage system.

Though SSD caching is vital for the IO performance, existing caching schemes face severe challenges in the virtualization environment. Firstly, existing caching schemes [5][6] mainly focus on critical performance metrics concerning one single VM, such as the IO latency, throughput, or the cache miss rate. However, transactional applications deployed in the Internet-based virtualization environment usually consist of multiple VMs with relationships among different nodes. When processing a request, multiple VMs across different cloud providers will be used, and the contribution to the response time for each VM is different. Ignoring the relationships may not lead to the application-level best performance, i.e., the response time. The key insight here is to utilize the application-induced relationships among VMs (often hosted on different cloud providers) to better guide the allocation of the SSD cache. Furthermore, as SSD has a significant advantage over HDD especially for random IO operations, allocating cache to VMs facing more random IO operations will potentially better utilize the scarce SSD cache. Secondly, transactional applications usually operate in a quite dynamic environment, i.e., the type of workloads is changing rapidly. The SSD caching scheme must be adaptive and intelligently so as to tune the cache allocation in response to the change of the workload. The key to designing an adaptive cache allocation scheme is to extract the meta-level invariance in or law behind the changes of the workload. Thus, it is indispensable to utilize the observation that the workload on different VMs often result from one single transactional application they belong to.

To address the challenges above, we present a novel SSD caching scheme named *Transaction-Aware SSD caching (TA-SSD)*. TA-SSD builds a MAPE-K[7] feedback loop to enable adaptive cache allocation. Using TA-SSD, transactional applications are instrumented to log the behavior and critical performance metrics at runtime. We then analyze the runtime log and performance data to figure out the workloads, the relationships among VMs, and the characteristics of low-level IO. Based on the application-level metrics, we use the genetic algorithm to obtain a nearly optimal cache allocation plan. The cache allocation plan is finally applied to the VMs.

We implement TA-SSD based on Xen[8] hypervisor, and evaluate it by using TPC-W[9][10] benchmark. TA-SSD can also be applied to other hypervisors such as KVM by modifying the execution part in the feedback loop. The evaluation shows that TA-SSD can improve the performance of the transactional application by up to 40% even when facing the

0-7695-6394-5/18/$31.00 ©2018 IEEE
DOI 10.1109/SOSE.2018.00029

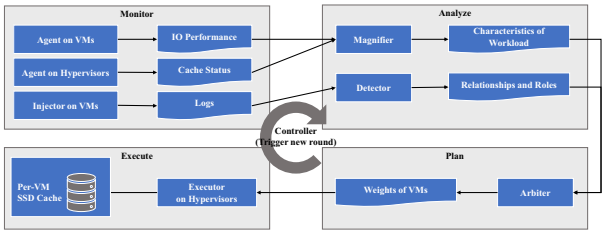


Fig. 1. Overview of TA-SSD

change of workload.

The rest of the paper is organized as follows. In section II, we discuss the overview of the tool TA-SSD, along with the adaptive closed loop we used. In section III, we give the formal definition of the problem and introduce the genetic algorithm based approach we used to solve it. In section IV, we use several experiments to show the self-adaptation and the performance improvement bringing by TA-SSD. We discuss the related work in section V. The section VI concludes the paper.

II. Overview of TA-SSD

In this section, we introduce the closed loop adaptation we use in TA-SSD, and the system architecture of TA-SSD. The overview of TA-SSD is shown in Fig. 1.

A. Motivation

The transactional application exposes a HTTP-based user interface or API, for user to access or build the open platform for third-party applications to connect. For transactional applications, the latency (time to process user requests) is the most significant performance metric. Reducing the average latency will result in better user experience. However, doing static SSD cache allocation to get optimized latency for transactional application is difficult. Firstly, when processing a user request, multiple VMs inside an application may be used. Different sets of VMs may be used when handling different user requests. Thus the relationships among VMs need to be dig out from the process of handling different types requests. We need to apply fine-grained monitoring to the VMs from multiple levels, so as to analyze the characteristics of the application and the low level IO. Secondly, considering that a transactional application usually has multiple workload types, the loads or access patterns of the application may change frequently. Moreover, the application may face burst load or sudden change of the access pattern in the runtime. This motivates us to introduce the closed loop adaptation into our approach, to react to the change of the environment quickly. Therefore, we need a flexible and adaptive SSD caching approach.

B. Overview of the Closed Loop Adaptation

We present the closed loop adaptation to adaptively allocate the SSD cache for VM from the view of the transactional application and trigger the new round after detecting the change of the workload. The closed loop is inspired by the MAPE-K feedback loop, consisting of the monitor, analyze, plan and execute process.

1) Monitor: We do the fine-grained monitor from multiple levels, including the behavior of the application, the IO performance of VMs and the cache status gathered from the Hypervisor. We instrument the target transactional application to record access logs of the execution time of the request along with the outgoing HTTP connections and database operations, so as to get the distribution of the latency over VMs. To specify the application which the VM belongs to, we deploy agents on each VM to monitor the running processes and TCP ports. We also deploy agents on hypervisors to monitor the status of cache.

2) Analyze: By using the monitored information on VMs, we then figure out the applications they belong to, the characteristics of the workload and the latency of the application. For each VM, we infer the application which it belongs to by analyzing the running processes, the opening ports, and the outgoing connections. Furthermore, we analyze all access logs and figure out the intensity of each type of the transaction. Also, for each transaction, we analyze the characteristics of low-level IO operations of each VM from the monitored IO performance data.

3) Plan: After monitoring the status of applications, hypervisors and VMs, and analyzing the relationships among VMs, the workload and the IO characteristics of VMs, we then calculate the SSD cache allocation plan for each application found in the cluster. Currently the total SSD cache size for a specific application is given, we need to further allocate the cache under the restriction that the total size is limited and the SSD on each hypervisor is limited. We use the genetic algorithm to solve the problem and try to get a nearly optimal solution as calculating an optimized solution in exponential time is unrealistic for the cloud provider. Finally, we output the weight of different VMs inside the application, which is used in the execute progress. More detailed information can be found in section III.

4) Execute: After calculating weight of different VMs inside an application, we allocate the SSD cache for each VM following the plan. The adjustment of SSD cache is done in hypervisor. Currently our approach is based on the Xen hypervisor[8]. Each cache is in the form of a file put in SSD and binds to a specific device in Xen hypervisor. The cache is based on the device mapper framework of Linux. We map the cache device and the mounted device of VM image into a cached device. We then use the target device with cache in the configuration of VMs in Xen hypervisor.

5) Trigger the New Loop: In general, we start the new round of cache adjustment frequently, for every minute, aiming to evolute for the specific workload mode. Furthermore, as the characteristics of workload is monitored continuously in the runtime, we immediately trigger a new round of cache adjustment after we detect the change in the mix of the transaction. Thus we can do the SSD cache allocation adaptively and can gracefully face the change of the environment.

C. System Architecture

TA-SSD consists of the following components: the centralized Controller, the Injector and the Agents for monitoring, the Detector and the Magnifier for analyzing, the Arbiter for planning and the Executor for executing, as shown in Fig. 1.

1) Controller: The centralized **Controller** triggers, traces and maintains the whole closed loop, and gives a uniformed API for management. The controller interacts with all the other components and gathers the information from them. It also traces the closed loop adaptation, and start the new round of cache adjustment frequently or after detecting the workload change.

2) Monitor: We deploy the **Injector** on the VM, and the **Agent** on both the VM and the hypervisor. The **Injector** is used to do instrumentation on target system so as to trace the behavior of the application. Currently we support the Java based web application. We instrument each servlet class, add code in the request handling method to record the execution time, and monitor the access of *URLConnection* and *JDBC* to get the distribution of the latency. The instrumented application will output the log upon every web interaction, the call of services on other VMs and the access of the database. The **Agent** deployed on the VM then gathers logs and monitors the running process on the VM along with the established TCP connections for further analysis. The **Agent** further monitors the detailed usage of CPU, including the user time and the IO time, to indicate the load of low-level IO on the VM. We also deploy the **Agent** on the hypervisor to monitor the cache status, including the cache utilization and miss rate of read and write operations.

3) Analyze: We present the **Detector** to figure out the VMs belong to the application, and the **Magnifier** to analyze the latency from the application level and the IO performance. The **Detector** infers roles of different VMs from the running processes gathered from the **Agent**. To infer the relationships among VMs, the **Detector** analyzes established TCP connections and matches them with logs of VMs, so as to figure out both the application and VMs involved for different transactions. The **Magnifier** focuses on analyzing the characteristics of the workload from logs, and the characteristics of IO operations from the performance data. It analyzes logs to find out the mix of the transaction in the current workload, and the distribution of the latency for each transaction.

4) Plan: We present the **Arbiter** to calculate the weight of each VM inside the application for the further cache allocation. The **Arbiter** is the fundamental component in TA-SSD, which uses the genetic algorithm to find out a nearly optimal solution consisting of weights of VMs inside the application. It uses the role of VMs, the relationships among VMs, the characteristics of the workload and IO operations, and the monitored performance data as the input, and evolutes to give an optimized SSD cache allocation plan. More detailed information can be found in III.

5) Execute: We deploy the **Executor** on the hypervisor. The **Executor** is responsible for allocating the cache according to the weights calculated by the **Arbiter**. Furthermore, it triggers the resize operation of caches for VMs inside the application. Currently, to change the size of the cache, we need to hang up the IO operations for a short time to resize the cache file for new configuration.

III. Nearly Optimal Planning based on Genetic Algorithm

In this section, we firstly formalize the problem and then introduce the genetic algorithm we used in TA-SSD.

A. Problem Definition

We regard the workload as a set of user requests.

Definition 1: **Workload.** The *Workload* is the two-tuple $W = (T, w)$, while $T = \{t_1, t_2, \ldots, t_n\}$ is the set of the transactions and $w = \{w_1, w_2, \ldots, w_n\}$ is the set of the ratio. We also have $\sum_{i=1}^{n} w_i = 1$.

We regard the application as a set of virtual machines, or what we call "application component". One application consists of several application components, with each of them deployed on different virtual machines.

Definition 2: **Application.** The *Application* is the set of n application components, $A = \{c_1, c_2, \ldots, c_n\}$.

The set A is extracted from the workload. Assuming that when handling the transaction t_i, the application components $c(t_i) = c_j, c_k, ..., c_l$ are used. We have $A = \bigcup_{i=1}^{n} c(t_i)$.

Using the definition of the workload and application, we can then induce the definition of the performance of the transactional application, which is the average latency. For a transaction, we have $L(t_i) = \sum_{j=k}^{l} L(T_i, c_j)$. Thus, the performance of transactional application can be defined as the latency $L = \sum_{i=1}^{n} L(t_i) \cdot w_i$. The goal of flash cache allocation is to get $\min(L)$.

Assuming that we allocate flash cache s_i for the application component c_i, with the miss rate mr_i. We have the IO latency of the VM when reading a data block of specific size $l = mr \cdot l_{HDD} + (1 - mr) \cdot l_{SSD}$, while l_{HDD} indicates the latency of HDD for reading the data block and l_{SSD} indicates the latency of SSD.

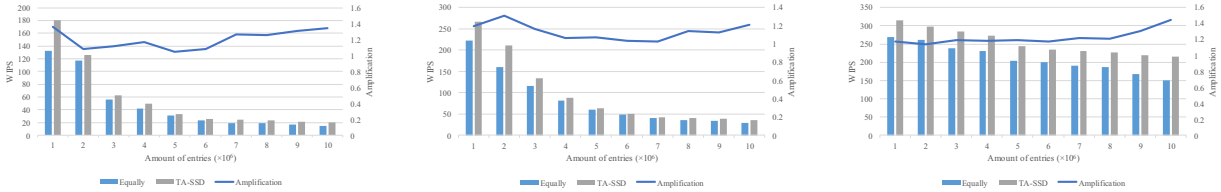
We can then estimate the latency on the VM as follows: $L(T_i, c_j) = \alpha \cdot (l_{ran} \cdot r_{ran} + l_{seq}(1 - \cdot r_{ran}))$. Noting that l_{ran} is the average latency for random access, while l_{seq} is that for sequential access. α indicates the dependency on the random access while handling the transaction T_i on the VM c_j, which can be estimated by comparing the IO time and the random access ratio of the workload and the transaction intensity. r_{ran} is the ratio of random access, which we monitored before.

Thus, to get the $\min(L)$, we aim to create connection between high-level latency of transactional applications to the low-level characteristics of IO operations, especially for dependency and random access.

B. Genetic Algorithm

Calculating an optimized solution in exponential time is unrealistic, thus we choose to use genetic algorithm to get a nearly optimal solution.

There are some critical parameters in the genetic algorithm which we need to introduce here. The **chromosome** indicates



(a) Browsing mode (b) Shopping mode (c) Ordering mode

Fig. 2. TPC-W WIPS

the weight of a specific VM, which is the ratio of the cache space. The **genome** indicates a specific SSD cache allocation plan, which consists of a set of chromosomes with the normalized (sum up to 1) weights of every VM inside an application. We define the **selection** operation as selecting genomes randomly by their fitnesses. We define the **crossover** operation as selecting random numbers of chromosomes of two genomes and swap them. We define the **mutation** operation as changing weights of random numbers of chromosomes of a genome inside a specific range.

For the most important part, the **fitness** aims to help predict the latency. We calculate the fitness from the following 3 metrics: the VM intensity, the IO ratio, and the random ratio. Firstly, the VM intensity is calculated by summing up the latency on specific VM among all transactions, which figures out the importance of the VM inside the application. Secondly, the IO ratio is the ratio of IO and non-IO operations, as optimizing a VM with low ratio of IO operations is not worthy, even if the VM is important. Thirdly, as VMs with high ratio of random access will benefit significantly from SSD cache, we takes the random ratio into consideration, which is the reciprocal of the average size of IO operations. Finally, we regard the fitness as the opposite number of the euclidean metric of the given weight with the three metrics.

IV. Evaluation

In this section, we evaluate TA-SSD and aim to answer the following research questions:

1) Can TA-SSD improve the performance of transactional applications?
2) Can TA-SSD adapt to the changing of workloads?

As we focus on the transactional application, we use TPC-W[9][10], a widely used e-commerce benchmark, in the evaluation. We use a TPC-W based load testing tool, Bench4Q[11], to generate the workload and evaluate the performance of the target system.

In the experiment environment, we deploy three applications on two hypervisors, to simulate the Internet-based virtual computing environment. Each hypervisor has two 8-core Intel Xeon CPU and 16GB memory. A 240GB Intel 535 SSD is attached, along with one Toshiba AL13SEB300 300GB SAS hard disk drive. The VMDKs are stored on HDD.

Currently for each application we build a TPC-W System Under Test (SUT) of two VMs. One is the Apache Tomcat web server, while the other is the MySQL database server. We put two VMs on different hypervisor to simulate the access across different cloud providers. Each VM is equipped with 2 vCPUs and 1GB of RAM, and 16GB of HDD.

We use 3 VMs to generate workloads and do load testing on 3 applications. Each VM is also equipped with 2 vCPUs and 1GB of RAM, and 16GB of HDD. Furthermore, for each application, we have a total cache size of 512MB available to allocate to VMs.

In the evaluation part, we compare the performance to the equally partitioned cache, i.e., to allocate a fixed size of the cache at a certain percentage of the HDD size of the VM. As there are 2 VMs in each application, we allocate 256MB of the SSD cache to each VM.

A. Performance

Firstly, we evaluate the performance from the view of one application and multiple applications.

1) Performance Focusing on One Application: We compare the performance of our approach with the equally partitioned cache in the three modes of TPC-W, which are BROWSING, SHOPPING and ORDERING. We change the mode of TPC-W benchmark with the based load of 100 concurrent virtual users and scale the data from 100,000 entries (the number of books) to 1,000,000 entries.

We use the TPC-W WIPS (Web Interactions Per Second) as the major performance metric as it indicates the throughput of the application. The comparison of WIPS under different modes is shown in Fig. 2(a) (for the browsing mode), 2(b) (for the shopping mode) and 2(c) (for the ordering mode).

We observed that the WIPS (Web Interactions Per Second) value is improved by up to 40% when using TA-SSD, comparing to the equally partitioned cache. The performance amplification increases stably and slowly when the data scale increases. Moreover, as the data scale increases, the working sets for the web server and database server also increase. This results in the performance degradation, especially for the workload type with high ratio of read operations, which explains the noticeably difference of Fig. 2(c) comparing to Fig. 2(a) and 2(b).

We also observed that the decrease of WIPS under the Ordering mode is not so much as that of the Browsing and Shopping mode. This is because the ratio of searching new books and viewing the detail of books is small, and such operations require high load of database scanning which causes

TABLE I
OVERVIEW OF TRANSACTIONS

Abbreviation	Description	Dominant IO
ADMC	Admin Confirm	(No Dominant IO)
ADMR	Admin Request	Random read
BESS	Best Seller	Sequential read
BUYC	Buy Confirm	Random write
BUYR	Buy Request	Random read
CREG	Registration	(No Dominant IO)
HOME	Home	Random read
NEWP	New Products	Sequential read
ORDD	Order Display	Random read
ORDI	Order Inquiry	(No Dominant IO)
PROD	Product Detail	Random read
SREQ	Search Request	(No Dominant IO)
SRES	Search Result	Sequential Read
SHOP	Shopping Cart	Random write

the cache replacement. Also, the improvement is significant for the Ordering mode, as it consists of high ratio of ordering operations, which triggers the random write operations to the database.

In summary, TA-SSD monitors and optimizes the performance of random read and write operations, so as it can handle this well.

To further figure out the efficiency of cache allocation, we analyze the performance improvement of different types of the transaction, which is shown in Fig. 3. The results is for the data scale of 1,000,000 entries.

We've analyzed the behavior of all transactions in the TPC-W SUT and list the dominant IO operations in table I. For the ADMC, CREG, ORDI and SREQ transactions, a simple form is transferred to the user, and only images for the web page are loaded, with no heavy IO requests operations. Moreover, for the BESS, NEWP and SRES transactions, a full table scan may be triggered, which results in the sequential read of the database.

We observe up to 50% decrease in latency, especially for those transaction types with a high ratio of random write operations, as SSD cache can benefit a lot from random operations and TA-SSD takes the characteristics of random IO operations into consideration. Moreover, we have found that for the NEWP transaction, TA-SSD gets almost similar latency comparing to the equally allocated cache (approximately 5% decrease of latency). This is because when processing the NEWP transaction, a full table scan is triggered to filter books of specific subjects, which is time consuming on both CPU and IO when the data scale is large.

2) Performance among All Applications: We also evaluate TA-SSD in the scenarios with multiple tenants. We deploy three TPC-W SUTs on two hypervisors. Each application has different workload mode and different data scale. The application 1 is populated with 300,000 entries, while the amount of entries for application 2 is 500,000 and for application 3 is 700,000.

We choose three different deployment plans for the 3 applications, which is deploying the CPU-sensitive VMs of

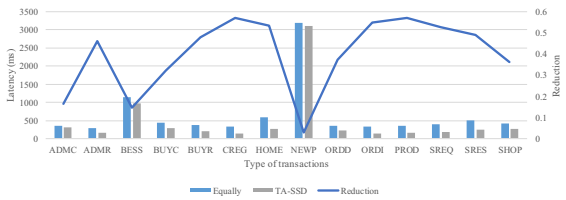


Fig. 3. TPC-W Latency of different transactions.

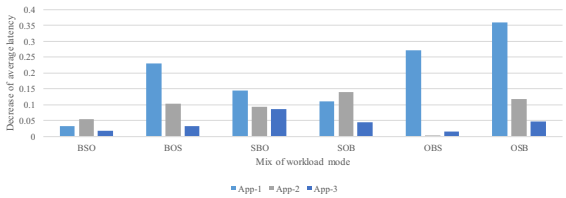


Fig. 4. Decrease of average latency. The letter "B" indicates the Browsing mode, while "S" for the Shopping mode and "O" for the Ordering mode.

2 applications together on the same hypervisor, deploying the applications over different hypervisors, and deploying the applications on the same hypervisor. Similar to the scenario of focusing on one application, the base load for each of the 3 application is 100.

We mix the three modes of TPC-W over three applications, confirming that each application are under different modes, and then apply the cache allocation on it. We still compare the WIPS to the equally partitioned cache. The results are shown in Fig. 4. The x axis indicates the modes of three applications. The y axis indicates the decrease of average latency of 3 applications, by comparing TA-SSD to the equally partitioned cache allocation.

We observe that TA-SSD can boost the performance of the individual application for up to 45% comparing to the equally partitioned cache, and with average performance improvement of 20%. Also, TA-SSD works well for applications with high ratio of random operations, even when multiple applications compete for the IO resources.

B. Self-Adaptation

We evaluate TA-SSD and find out if it can handle the change of the workload gracefully. We still compare the performance of our approach and the equally partitioned cache. It can be assumed as a straight-forward approach in self adaptation, as the data blocks in the cache will be replace to adapt to new environment.

1) Change of Base Load: Firstly, we change the base workload of TPC-W benchmark, and find out if TA-SSD can handle this. We focus on one application consists of 2 VMs deployed on different hypervisors. The data scale is 500,000 and the workload mode is Ordering. We perform the experiment lasting for 4 minutes. The base load is 50 for the first minute, 100 for the second minute, 200 for the third minute and 150 for the forth minute, to simulate different overheads.

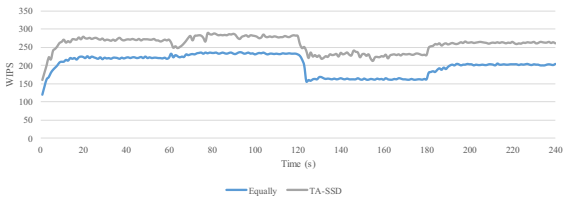


Fig. 5. Change of base load

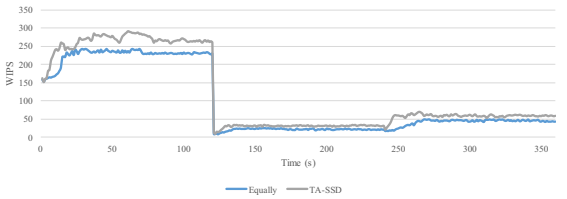


Fig. 6. Change of workload type

The results are shown in Fig. 5. We observe that comparing to the equally partitioned cache, TA-SSD can adapt to the new workload more quickly. Moreover, for the stabilized performance, TA-SSD is up to 40% better than equally partitioned cache. Also, we observe from the figure that for the extreme heavy load (200 base load), the WIPS value for both approach is decreased significantly. This is due to large number of failed requests marked by the load generator, which is timeout (longer than the tolerable wait time for the user) or error.

2) Change of Workload Type: We change the type of the workload and compare the performance of TA-SSD and equally partitioned cache. We change the type of the workload for every two minutes and find out if TA-SSD can quickly react to the new workload. We focus on one application consists of 2 VMs deployed on different hypervisors. The data scale is 500,000 and the base load is 100. We perform the experiment lasting for 6 minutes, and change the workload mode by the order of Ordering, Browsing and Shopping, and each lasts for 2 minutes. The results are shown in Fig. 6. We observe that TA-SSD can quickly react to the workload change, comparing to the equally partitioned cache. Thus, with the help of closed loop adaptation, TA-SSD is able to react to the change of the environment quickly and efficiently.

V. Related Work

In this section, we focus on the related work for the resource management of Flash-based SSD and cache. **Centaur**[6] uses curves of miss rate and IO latency to help allocate the per-VM SSD cache and meet the requirement of VMs. **S-CAVE**[12] considers the number of reused blocks to help identify the cache space demand of each VM. Unlike Centaur and S-CAVE, TA-SSD not only uses the low-level performance data, but also analyzes the characteristics of transaction applications to help allocate the SSD cache. **CloudCache**[13] proposes a new cache demand model for SSD cache, using Reuse Working Set (RWS) to satisfy the requirement of workloads. Unlike CloudCache, TA-SSD creates connections between the high-level application and the low-level IO performance to figure out the SSD cache requirement of VMs.

VI. Conclusion

We present TA-SSD, a novel tool to allocate SSD cache in Internet-based virtualization environment, in order to improve the performance of the transactional application. We introduced the MAPE-K feedback loop to automatically detect the change of the workload mode and trigger the cache adjustment. Moreover, we use the genetic algorithm based SSD cache allocation approach to calculate a nearly optimal solution quickly and efficiently. The evaluation shows that TA-SSD has the significant advantage over the equally partitioned cache.

Acknowledgment

This work was supported by the National Key Research and Development Program of China (2016YFB1000103), the National Natural Science Foundation of China (61572480), Tianjin Massive Data Processing Technology Laboratory, and Youth Innovation Promotion Association, CAS (No. 2015088).

References

[1] H. Wang, P. Shi, and Y. Zhang, "Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 1846–1855.

[2] B. An, X. Shan, Z. Cui, C. Cao, and D. Cao, "Workspace as a service: An online working environment for private cloud," in *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, April 2017, pp. 19–27.

[3] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST'16. USENIX Association, 2016, pp. 133–148.

[4] J. Kim, D. Lee, and S. H. Noh, "Towards slo complying ssds through ops isolation," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15. USENIX Association, 2015, pp. 183–189.

[5] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer, "Mercury: Host-side flash caching for the data center," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, April 2012, pp. 1–12.

[6] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side ssd caching for storage performance control," in *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, July 2015, pp. 51–60.

[7] A. Computing *et al.*, "An architectural blueprint for autonomic computing," *IBM White Paper*, 2006.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. ACM, 2003, pp. 164–177.

[9] D. Menasce *et al.*, "Tpc-w: A benchmark for e-commerce," *Internet Computing, IEEE*, vol. 6, no. 3, pp. 83–87, 2002.

[10] D. F. García and J. García, "Tpc-w e-commerce benchmark evaluation," *Computer*, vol. 36, no. 2, pp. 42–48, 2003.

[11] W. Zhang, S. Wang, W. Wang, and H. Zhong, "Bench4q: a qos-oriented e-commerce benchmark," in *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*. IEEE, 2011, pp. 38–47.

[12] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou, "S-cave: Effective ssd caching to improve virtual machine storage performance," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13. IEEE Press, 2013, pp. 103–112.

[13] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao, "Cloudcache: On-demand flash cache management for cloud computing," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST'16. USENIX Association, 2016, pp. 355–369.