Exploiting concurrency to improve latency and throughput in a hybrid storage system

Xiaojian Wu, A. L. Narasimha Reddy
Department of Electrical and Computer Engineering, Texas A&M University
College Station, Texas 77843
Email: tristan.woo@neo.tamu.edu, reddy@ece.tamu.edu

***Abstract*—This paper considers the problem of how to improve the performance of hybrid storage system employing solid state disks and hard disk drives. We utilize both initial block allocation as well as migration to reach "Wardrop equilibrium", in which the response times of different devices equalize. We show that such a policy allows adaptive load balancing across devices of different performance. We also show that such a policy exploits parallelism in the storage system effectively to improve throughput and latency simultaneously. We implemented a prototype in Linux and evaluated it in multiple workloads and multiple configurations. The results show that the proposed approach improved both the latency of requests and the throughput significantly, and it adapted to different configurations of the system under different workloads.**

I. INTRODUCTION

As faster Solid State Devices (SSDs) become available, there is an increasing interest in utilizing these devices to improve the storage system's performance. Many organizations and architectures are being pursued in employing the SSDs in storage systems. In this paper, we consider the hybrid storage systems that employ both SSDs and magnetic disk drives together in one system.

Traditionally, memory systems and storage systems employed a hierarchy to exploit the locality of data accesses. In such systems, data is cached and accessed from the faster devices while the slower devices provide data to the faster devices when data access results in a miss at the faster devices. Data is moved between the different layers of the storage hierarchy based on the data access characteristics.

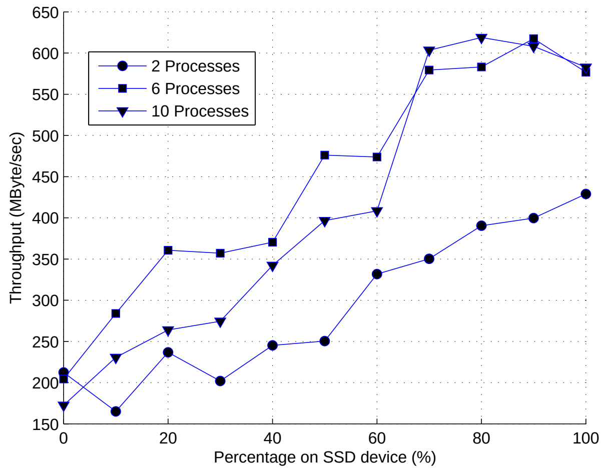
Employing faster devices as caches generally improves performance while hiding the complexity of handling the diversity of multiple devices with different characteristics to the upper layers. However, caching generally results only in realizing the capacity of the larger (and slower) devices since the capacity of the faster (and smaller) devices is not exposed to the upper layers.

When the capacity of the devices at different layers can be comparable, it is possible to employ other organizations to realize the combined capacity of the devices. Migration is one of the techniques employed in such situations. In such systems, the frequently or recently accessed data is stored on (or migrated to) the faster devices to improve performance while realizing the combined capacity of the devices. Even when migration is employed, the storage system can still be organized as a hierarchy with faster devices at higher layers being accessed first on any data access.

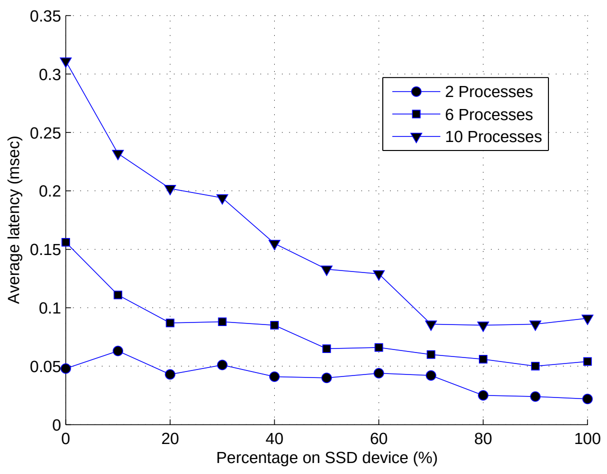
When the storage system is organized as a hierarchy (either with caching or migration), the data throughput can be limited by the rate at which data can be accessed from the faster device (even when all the data is accessed from the faster devices). The throughput is limited by the fact that the faster device(s) has to be accessed on every access. However, it is possible to provide higher throughput if data can be accessed directly from all the devices without enforcing a strict hierarchy. In such a system, it is possible to provide throughput higher than what can be provided by a strictly hierarchical system.

We explore this option, in this paper, in organizing SSDs and magnetic disks in a hybrid system. This is driven partly by the fact that while the small read (or random read) performance of SSDs can be significantly higher than magnetic disks, the large read/write access performance of SSDs is comparable to magnetic disks and random write performance of SSDs can be sometimes worse than that of magnetic disks, depending on the choice of SSDs and magnetic disks. Second, depending on the prices of storage on different devices, systems will be designed with different amounts of storage on SSDs and magnetic disks. We are motivated to design a storage organization that works well across many different organizations with various levels of SSD storage in a hybrid storage system.

Other systems have explored the organization of storage systems in a non-hierarchical fashion, for example [3]. The data can be stored in either type (faster or slower) device and accessed in parallel in such systems. For example, on a miss, read data may be directly returned from the slower device while making a copy in the faster device, in the background, for future references. Similarly, on a write miss, the writes may be directly written to the slower device without moving data to the faster device. However, most such systems employ faster devices until their capacity is exhausted before they start moving data to the slower devices. In the system, we explore here, the capacity of the faster device may not be completely utilized, before data is allocated on the slower devices and used in parallel with the faster devices. Our approach tries to improve both the average latency and throughput of data access simultaneously by exploiting the parallelism that is



(a)



(b)

Fig. 1. Performance of hybrid device with static allocation ratios. (Using dbench.) (a) Throughput. (b) Average latency.

possible through accessing multiple devices concurrently.

When data is striped across multiple devices, data can be accessed in parallel from all the devices, potentially making use of all the devices in the system. However, striping allocates data uniformly across all the devices without regard to the relative performance of the devices. In the approach we adopt here, the data is distributed differently across the different devices based on the observed performance of the devices and thus tries to match load across the devices to dynamically observed performance of the devices.

In order to accommodate the diversity of the devices (both SSDs and magnetic disks) and the continuous evolution of the SSD architectures and characteristics, we employ a completely performance-driven approach to this problem. We measure the performance of the devices for various requests and use the measured characteristics to drive our data organization across the multiple devices in the hybrid storage system. Our approach hence adjusts to the workloads and the configuration of number of different devices in the system. We will show through results, based on two workloads and multiple configurations of storage systems, that our approach improves performance, compared to a strict-hierarchical approach.

The paper makes the following significant contributions: (a) designs a space allocation policy that can exploit parallelism across the devices while providing low average latencies, (b) designs a policy that adjusts to the workloads and to different configurations in a hybrid system and (c) evaluates the proposed policy on a Linux testbed using synthetic benchmarks and real-world traces to show that the proposed policy achieves its goals.

The remainder of this paper is organized as follows. Section II describes the proposed approach to managing the space in a hybrid system, and our prototype implementation. Section III presents the results of the evaluation. Section IV gives details of the related work. Section V concludes the paper.

II. Design and Implementation

Our design is motivated by the results shown in Figure 1. We considered a system with one Intel SSD 2.5 inch X25-M 80GB SATA SSD and one 15K RPM, 73G Fujitsu SCSI disk. We ran a workload of dbench benchmark [4] on a storage system employing these two devices in various configurations. We considered a system where all the data is stored entirely on the SSD and different levels of allocation across the two devices. For example, in a system employing an 80/20 allocation, 80% of the data is allocated on the SSD and 20% of the data is allocated on the hard disk. The benchmark we considered is small enough such that the entire dataset can fit on the SSD. Hence, the results shown in Figure 1 exclude the overheads of managing the data across the two devices in various configurations. It is seen that the performance is not necessarily maximized by allocating all the data on the faster device, in this case the Intel SSD.

The dbench benchmark uses throughput as a performance measure and throughput, in this case, is clearly improved when the data is stored across both the devices, enabling both the devices to be used in parallel. We also measured the average latency of all the NFS operations in the benchmark. It is observed that the throughput and latency of the system continues to improve as more and more data is allocated on the SSD, until 100% of the data is allocated on SSD, for the two workloads of 2 and 6 processes. However, with 10 processes, the throughput is maximized at about 80% allocation (on SSD) and latency is minimized at about 70% allocation.

The results in Figure 1 also clearly show the impact of a strict hierarchical access of data. As we increase the number of processes, the SSD performance peaked at about 620MB/s and at higher number of processes, organizations that employ both SSD and disk in parallel can do better. This is an important observation that leads to our design below. When the amount of parallelism in the request stream is low, a hierarchical design works well. However, as the amount of parallelism

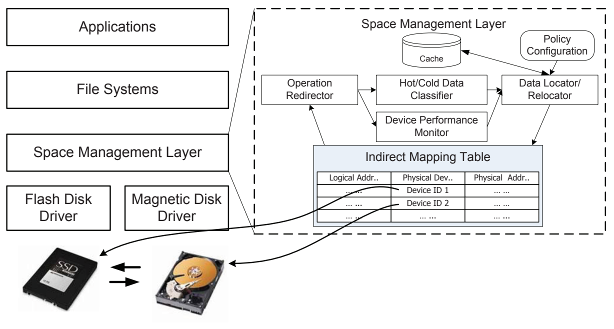


Fig. 2. Architecture of space management layer.

increases in the request stream, the requests can be better served by exploiting multiple devices, even at the cost of employing slower devices in serving some of the requests. The reduction in waiting times to serve requests can contribute to improvement in performance.

It is emphasized that our experiment here employed static allocations of different percentages across the devices to study the feasibility of doing better than a strict hierarchical system and our results do not include the costs of managing the data across the two devices (in all configurations). However, these results point to the potential of exploiting the parallelism in the storage system through concurrent data access from multiple devices.

These results motivate the work in this paper. We design a system that can utilize all the devices in the hybrid storage system irrespective of the number of devices in the storage system. We design a system that automatically adjusts the allocation to different types of devices to improve the performance of the whole system. The allocation percentages are automatically adjusted based on the number of different types of devices in the system. We implemented a Linux prototype employing our ideas. We show through trace-driven evaluation of our system, that our system can improve throughput and latency of data access simultaneously compared to a strictly hierarchical system.

A. Allocation and Migration

We implemented a prototype in Linux OS to evaluate our ideas. The architecture of the prototype is shown in Figure 2. We designed a space management layer to allocate data between different devices in the system. The space management layer sits between the file system and the device diver and manages allocation of new blocks across the different devices along with migrating different blocks to different devices as the capacity and performance considerations dictate as explained below.

We take a block level approach so that the solution can work with any file system above. Measuring device level performance at the file system is made difficult due to page cache buffering and read ahead policies of the operating system.

The space management layer provides an indirection between the physical device addresses employed by the file system and the actual physical blocks on the device where data is written. It maintains a block map of where a given block is located on which device. We manage space in extents of 128K bytes. For each logical extent, we use one bit to indicate the underlying device and store a physical extent number where this extent can be found on the device. The overhead introduced by the mapping table is low. For example, for a system whose capacity is 1T bytes, the size of the full mapping table is $(1T/128K) * \lceil(\log_2(1T/128K) * 2 + 1)/8\rceil = 48M$ bytes, and the overhead is about 0.0046%. The space management layer also allocates new blocks. The allocation ratios and the migration rates from one device to another are controlled by the observed performance of the devices, always preferring the devices with faster performance. The device performance monitor in this layer keeps track of the request response times at different devices. And the hot/cold data classifier determines if a block should be considered hot. The cold data will only be migrated in the background.

The space management layer may allocate blocks on magnetic disks even before the space on the SSDs is completely allocated, depending on the observed performance of the SSDs relative to the magnetic disks.

This architecture supports flexible policy configuration. In our earlier work [32], we implemented a policy that tries to match read/write requests to the observed read/write access characteristics of different devices in a hybrid storage system. That policy uses migration as the main method to balance the workload and improves the performance significantly only when the underlying devices exhibit asymmetric read/write access characteristics. In this paper, we propose a new policy that only considers the aggregate response time as a performance metric and uses allocation as the main vehicle to balance the workload. As we will show, this policy can improve both throughput and latency even when the SSD device performs beyond the HDD device on both read as well as write operations. In the rest of this section, we describe the details of the policy.

Our basic approach is driven by a goal of trying to reach "Wardrop equilibrium". When the devices in the system are at a Wardrop equilibrium, the average response time for a data item cannot be improved by moving that data item alone from its current device to another device. Typically, Wardrop equilibrium can be reached when the response times of different devices equalize. Such an approach is employed in road transportation problems and in multi-path routing in networks [1], [5]. In order to equalize the response times of devices, we may have to subject the devices to different loads.

In this paper, we consider device performance in making initial storage allocations as well as migration of blocks after they are allocated. It is emphasized that these allocations can span across all the devices before the faster device's capacity is filled. In the current system, whenever a new block is written to the storage system, the storage system dynamically decides where to store that block of data based on observed performance of the devices. A space management layer keeps track of the location of the different blocks across the devices in the storage system, providing a mapping of file system logical block addresses to physical block addresses in the storage system. In order to keep this space management layer efficient, data is managed in "extents", different from file system blocks. An extent, typically, consists of multiple file system blocks.

This allocation policy tends to allocate more blocks on better performing devices and exploits slower devices as the faster devices get loaded with increasing request rates. As a result of this dynamic performance-driven approach, initially the data is allocated on the faster devices, SSD in our system. As the waiting times increase at the SSDs, the performance of a data access gets worse and at some point in time, data can be potentially accessed faster at an idle magnetic disk. When that happens, data is allocated on the magnetic disk as well, thus allowing parallelism to be exploited in serving requests. The details of this allocation policy are explained below, including how performance is measured, how allocation is shifted based on observed performance etc.

Blocks of data are migrated from one device to another based on several considerations. Cold data is moved to the larger, slower devices in the background, during idle times of the devices. Hot data could also be migrated when the loads on the devices are imbalanced or current device is not providing as good a performance to this block of data as it is likely to receive at another device. Migration happens asynchronous to the arriving request stream so that the delays in migration do not affect the request completion time. In order to ensure that migration of data doesn't impact the performance of the foreground user requests, we give migration requests lower priority than the current user requests. Migration can take three forms in our system. First, cold data is migrated, in the background, from faster device to larger devices to make room on the smaller, faster devices. Second, hot data is migrated to lighter-loaded devices, in the background, to even out performance of the devices. Third, data is migrated on writes, when writes are targeted to the heavier-loaded device and the memory cache has all the blocks of the extent to which the write blocks belong (explained further below).

Among the two options of adjusting load across the devices, allocation is simpler than migration. Data can be allocated on the currently fastest device or least-lightly loaded device. As a result of an allocation decision, we impact only one device. Migration impacts two devices, the device where the data currently resides and the new device where the data is migrated to. Ideally, data is moved from the most heavily loaded device to the least lightly loaded device. We make use of both the mechanisms to balance load and to exploit parallelism in the system. We employ migration when the current allocation rate is not expected to be sufficient to balance the load and when the smaller device starts filling up. In the first case, hot blocks are migrated from higher-loaded devices to lighter-loaded devices and in the second case, data is migrated from devices that are filling up to the devices that still have unfilled capacity (and when there is a choice among these devices, to the lighter-loaded device).

$$
\begin{aligned}
&\bar{R} = R_s \times Q_s + R_h \times Q_h \\
&P_s = Q_s + \alpha[\bar{R} - R_s] \times Q_s \\
&P_h = Q_h + \alpha[\bar{R} - R_h] \times Q_h \\
&TokenNumbers = |P_s - Q_s| \times \beta \\
&Direction = \begin{cases} \text{HD}->\text{SSD}, & \text{if } P_s > Q_s; \\ \text{SSD}->\text{HD}, & \text{else;} \end{cases}
\end{aligned}
\tag{1}
$$

We use a timer routine to track the performance of each device and the actual percentage of workload on each device. This routine also use this information to calculate the number of load-balancing tokens and the migration direction as specified in equation (1). We explain our mechanism in relation to two devices SSD and HD here to make things easier to understand. In equation (1), R_s and R_h are measured response times on both devices, Q_s and Q_h are measured workload distribution on each device, P_s and P_h are the target distribution we want to reach in the next round, and α and β are design parameters. If the measured response time of the device is worse(better) than the average response time in the system, the workload on that device is reduced (increased). In order to ensure that the performance of the devices can be measured, the allocations are lower bounded for each device (i.e., they can't be zero). We employ exponential averaging of response times to smooth out instantaneous bursts.

The load-balancing tokens are consumed by both allocation and migration. The number of tokens control the rate at which the load is adjusted across the devices. As explained earlier, allocation is a preferred mechanism for achieving our equilibrium goals. The load balancing tokens are used first by the allocation process. Depending on the values in the equations above, the allocation is slowly tilted towards the lighter-loaded device. The migration thread will only do migration when the load-balancing tokens are not completely consumed by the allocation process (for example, because there are no new writes). When a write request to new data arrives, the new data will be allocated to lighter-loaded device and consume one token if there is any. If there is no token available, the new data will be allocated according to the distribution P_s/P_h. A natural question that may come to mind is what happens to our system when the storage system is completely filled once. How does an allocation policy help in managing the load across the storage system? SSDs employ a copy-on-write policy because of the need for erasing the blocks before a write. In order for SSDs to carry out the erase operations efficiently, it is necessary for SSDs to know which blocks can be potentially erased (ahead of time, in the background) such that sufficient number of free blocks are always available for writing data efficiently. To accommodate

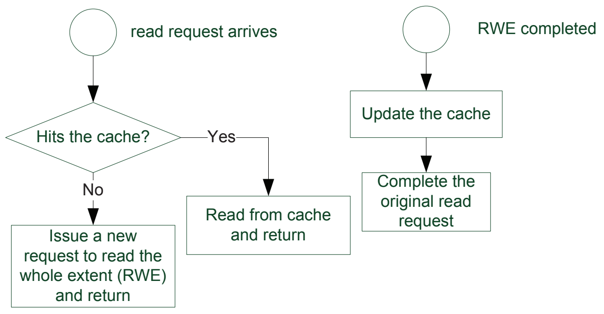

read request arrives
Hits the cache?
Yes
No
Read from cache and return
Issue a new request to read the whole extent (RWE) and return
RWE completed
Update the cache
Complete the original read request

Fig. 3. Handling the read requests.

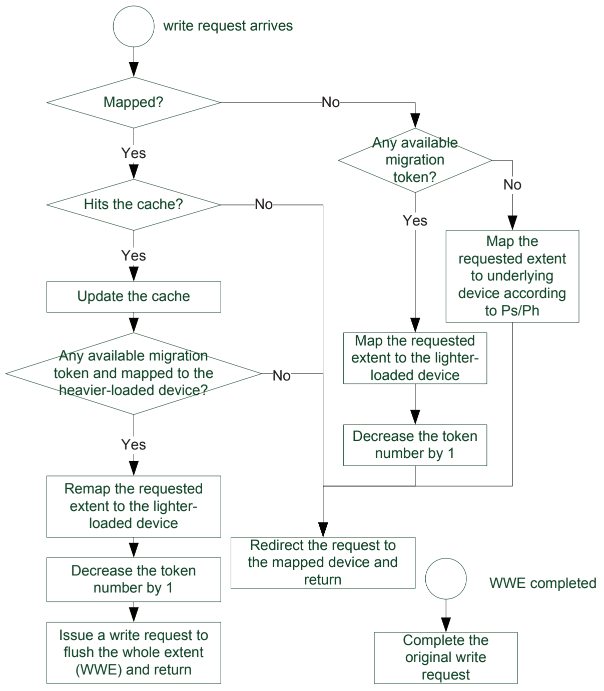

write request arrives
Mapped?
No
Yes
Any available migration token?
No
Yes
Hits the cache?
No
Yes
Map the requested extent to underlying device according to Ps/Ph
Update the cache
Map the requested extent to the lighter-loaded device
Any available migration token and mapped to the heavier-loaded device?
No
Decrease the token number by 1
Yes
Remap the requested extent to the lighter-loaded device
Redirect the request to the mapped device and return
WWE completed
Decrease the token number by 1
Issue a write request to flush the whole extent (WWE) and return
Complete the original write request

Fig. 4. Handling the write requests.

this need, TRIM ATA command has been developed [6]. This command allows a file system to communicate to the storage system which of the blocks on the storage system are not live blocks in the file system. We exploit this new command in developing our system.

When TRIM commands are issued, the storage system can keep track of available free space on different devices and continue to employ allocation decisions in guiding its policy of balancing load and performance of different devices. A write to a logical block that was previously allocated and "trimmed" can be considered as a new allocation. With such an approach, allocation can continue being a useful vehicle for balancing load across the devices until the system is completely full.

In this paper, in addition to the performance-driven migration and performance-driven allocation decisions as explained above, we also employed a number of techniques to improve performance. These include caching of entire extents in memory even if only a few of the blocks in the extent are read. This full-extent caching is expected to help during migration of blocks. Without such full-block caching, a hot block that requires migration from one device to another device may require several operations: reading of remaining blocks from the older device, merging of recently written/accessed data with the older blocks and migration of the entire extent of blocks to the new device. When the entire extent is read and cached, migration requires only the last operation since all the blocks of the extent are currently in memory.

We implement the cache as a write-through cache with pre-fetch mechanism as shown in Figures 3 and 4. When a read request arrives, if it hits the cache, it will be read from the cache and returned immediately. Otherwise, a new request (RWE) is issued to fetch the whole extent that contains the requested data. When this request is completed, the whole extent will be inserted into the cache and the original read request is returned to the user process. When a write request arrives, if the requested address is not mapped (i.e, write to a new extent), the requested address will be mapped to an underlying device. If there is any migration token waiting, it will be mapped to the lighter-loaded device, otherwise, the request is allocated according to the targeted distribution P_s/P_h in equation 1. If the requested address is mapped and the request misses in the cache, it will be redirected to the mapped device according to the mapping table. If the request hits in the cache and the requested address is mapped to the heavier-loaded device, the request extent might be re-mapped to the lighter-loaded device when there is an available token. We only need to flush the updated cache to the new mapped device to complete the migration from the old mapped device. As described above, the cache is write-through, which prevents any data loss during exceptions such as power failure. It is noted that caching here refers to memory caching and not caching in SSD.

We employ a cache of 100 recently read extents in memory. This cache is employed in all the configurations studied in this paper to keep the comparisons fair.

We compare our system against two other systems, one in which the capacity on the SSD is allocated first and a second system that stripes data across the two devices. The first system employs caching at SSD when the capacity of the SSD is exhausted. In the current system, the allocation across the devices is driven by dynamic performance metrics in the system and is not decided ahead of time.

III. Evaluation

A. Testbed Details

In the experimental environment, the test machine is a commodity PC system equipped with a 2.33GHz Intel Core2 Quad Processor Q8200, 1GB of main memory. The magnetic disks used in the experiments are 15K RPM, 73G Fujitsu SCSI disks (MBA3073NP), the flash disk is one Intel SSD 2.5 inch X25-M 80GB SATA SSD (SSDSA2MH080G1GC). The operating system used is Fedora 9 with a 2.6.28 kernel,

TABLE I
NUMBERS OF OPERATIONS BY TYPE IN ONE PROCESS OF DIFFERENT WORKLOADS

	dbench	NetApp Traces
NTCreateX	3390740	46614
Close	2490718	46610
Rename	143586	145
Unlink	684762	29
Deltree	86	0
Mkdir	43	0
Qpathinfo	3073335	99896
Qfileinfo	538596	22506
Qfsinfo	563566	37255
Sfileinfo	276214	50527
Find	1188248	34393
WriteX	1690616	47626
ReadX	5315377	43044
LockX	11042	0
UnlockX	11042	0
Flush	237654	118

TABLE II
AVERAGE SIZES OF READX/WRITEX OPERATIONS IN DIFFERENT WORKLOADS

	dbench	NetApp Trace
Average ReadX size (bytes)	11853	18105
Average WriteX size (bytes)	25720	6204

and the file system used is the Ext2 file system. We cleaned up the file system before each experiment.

B. Workloads

Dbench [4] is a filesystem benchmark using a single program to simulate the workload of the commercial Netbench benchmark. This benchmark reports both the latency of each NFS operation and the total throughput as metrics. To evaluate our policy extensively, we also use dbench to replay some traces from real world. The real traces were obtained from an engineering department at NetApp [11]. We developed a tool to translate the NetApp traces into the format that dbench can use. The tool generates two dbench trace files from each NetApp trace file. One of them is only used to initialize the file system image, the other one contains the exact same operation sequence as that in the original trace file. The workload is replicated under different directories as we increase the number of processes such that each process replays a workload of the original trace, different and independent of other processes.

The characteristics of request sizes and read/write ratios for the two workloads are shown in Table I and II. As seen from the table, the NetApp trace has smaller read/write ratio and smaller write request sizes.

In all the experiments, we take the design parameters $\delta = 10/ms$ and $\beta = 100$.

C. Results

Results of our experiments on a storage system with one SSD and one magnetic disk (called 1SSD+1HD configuration here) in a workload of dbench benchmark are shown in Figure 5, 6 and 7. Each experiment was run 5 times and the averages are shown. The 95% confidence bars were computed for all simulations, but not necessarily shown in the figure, to make it easier to read the data. The figures show a comparison of throughput (MB/s and IOPS/s) and latency across four systems, entirely running on the SSD, entirely running on the HD, on a hybrid system employing our policy and on a hybrid system employing a static allocation mix of 80/20 on SSD/HD (as identified earlier to be a good static configuration in Figure 1). The two configurations 70/30 and 80/20 performed similarly and we use the 80/20 configuration as an example. It is observed that our policy does nearly as well or better than the 80/20 static configuration and achieves higher throughput than the SSD or HD alone at higher number of processes. This shows that our policy dynamically adapted the allocation rate to individual devices and the workload to achieve good performance.

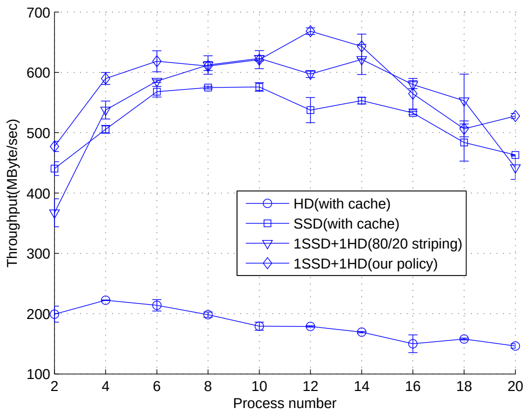


Fig. 5. Throughput of dbench workload in different configurations (higher is better)

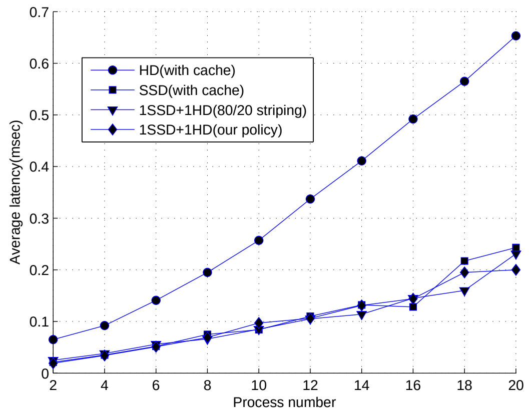


Fig. 6. Latency of requests in dbench workload (lower is better)

Results of our experiments on the storage system with 1SSD+1HD configuration in a workload derived from the NetApp traces are shown in Figures 8,9 and 10. The figures show

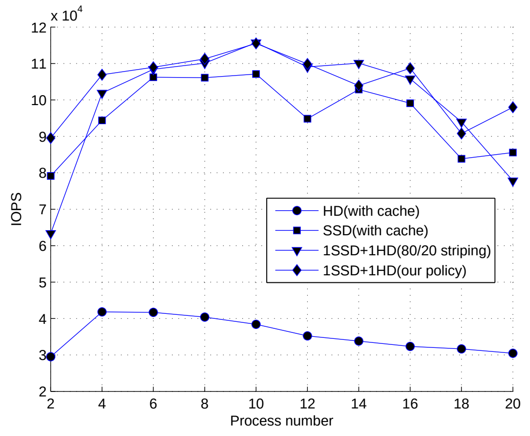


Fig. 7. IOPS in dbench workload (higher is better)

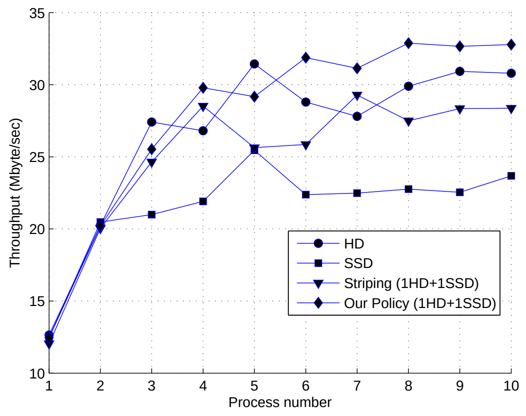


Fig. 8. Throughput of NetApp workload in different configurations (higher is better)

various performance measures as the number of processes is increased in the workload. It is observed that as the number of processes is increased, the allocation-based policy achieves higher throughput than when the data is entirely stored on the SSD or when the data is striped across the two devices (SSD and hard disk). The allocation-based policy achieves nearly 38% more throughput than the SSD and nearly 16% more throughput than a striping configuration, with 10 requesting processes. The NetApp workload has more write requests than read requests and the write requests are smaller. Both these characteristics contribute to the fact that for this workload,the throughput performance of magnetic disk drive is better than that compared to the SSD.

Figure 9 shows the average latency time for I/O requests. It is observed that the allocation-based policy simultaneously improves latency along with throughput. The allocation-based policy achieves nearly 28% better latency than the SSD and 17% better latency than the striping configuration, with 10 requesting processes. This is primarily due to the simultaneous use of both the devices and the appropriately proportional use of the devices based on their performance. As seen from the two workloads above, the allocation-based policy works well across two different workloads.

Below, we show that the proposed allocation-based policy works well in different configurations. We create additional configurations, 1SSD+2HD (1 Intel SSD and 2 Fujitsu magnetic disks) and 1SSD+4HD (1 Intel SSD and 4 Fujitsu magnetic disks). We study the performance of all the previously considered policies now in these new configurations. For the data on HD alone policy, we use the magnetic disks in a RAID0 configuration (striping with no parity protection). The results with dbench workload are shown in Figures 11 and 12. To make the figures clearer, we only plot the result of 1SSD+4HD. The performance of 1SSD+2HD is between those of 1SSD+1HD and 1SSD+4HD. It is observed that our allocation policy improved performance in the new 1SSD+4HD configuration, from 1SSD+1HD configuration. This increase

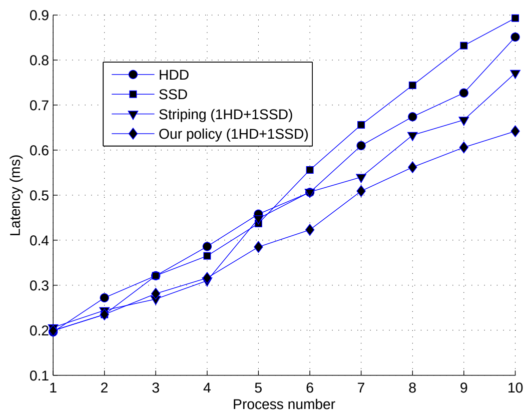


Fig. 9. Average latency of requests in NetApp workload (lower is better)

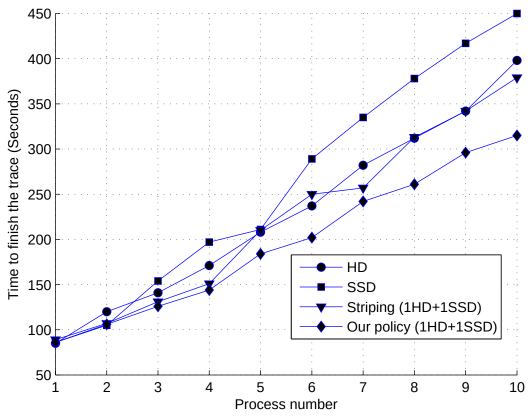


Fig. 10. Used time to finish the NetApp workload in different configurations (lower is better)

in performance comes from increased use of magnetic disks in supporting the workload. The performance of our policy is better than striping data across all the five devices (1SSD and 4 HDs) as shown in the figure. It was already shown earlier that our policy achieved better performance than when all the data resides on the single SSD. The data in Fig. 11 compares our policy with striping data on the four disks or when data is statically distributed in a 80/20 ratio across the SSDs and the four disks (the static allocation ratio that was found to work well earlier in the 1+1 configuration). The results show that our policy outperforms these other alternative options. The

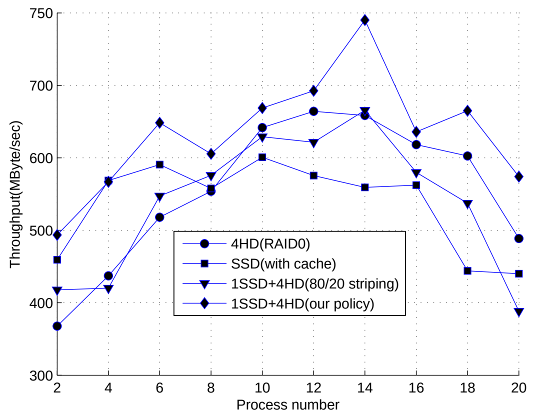


Fig. 11. Throughput of dbench workload in different 1+4 configurations (higher is better)

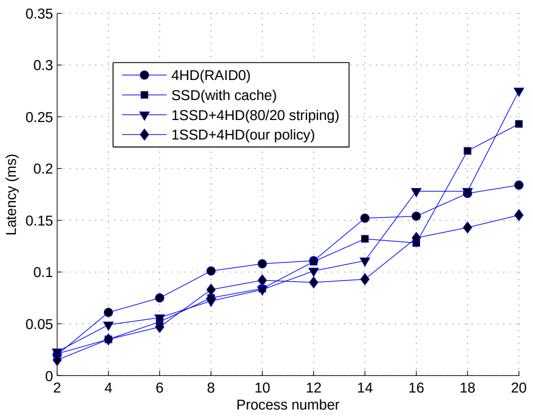


Fig. 12. Average latency of dbench requests in different 1+4 configurations (lower is better)

allocation percentages across the devices in these experiments, with the dbench workload, are shown in Figure 13. First, we observe that our policy allocated about 65-70% of data on the SSD in the 1+1 configuration. This allocation percentage is close to one of the better configurations, 70/30, identified earlier through static allocation. We can also see that more data is allocated to magnetic disks in the 1SSD+4HD configuration than in the 1+1 configuration. Our policy adopted allocations to the availability of more disks in the new configuration and allocated smaller amount of data to SSDs.

We also present the allocation percentages across SSD and HD for the NetApp workload in Fig. 14. It is observed that our policy distributed about 50% of the data to SSD and 50% of the data to the hard disk in this workload. As noticed earlier, the hard disk performs better in this workload than in the dbench workload. This is the reason more data is allocated to hard disk in this workload.

As the number of processes increases, in both workloads, the allocation percentage is shifted slightly more towards hard disk compared to the distribution at lower number of processes. With higher number of processes, more devices can be exploited, even if they are slower, to improve throughput. It is observed that the allocation percentage is slightly more than 50% on the hard disk, at higher number of processes, in the NetApp workload, as seen in Fig. 14. This is consistent with the higher performance of the hard disk compared to SSD in this workload, at higher concurrency, as seen earlier in Fig. 8.

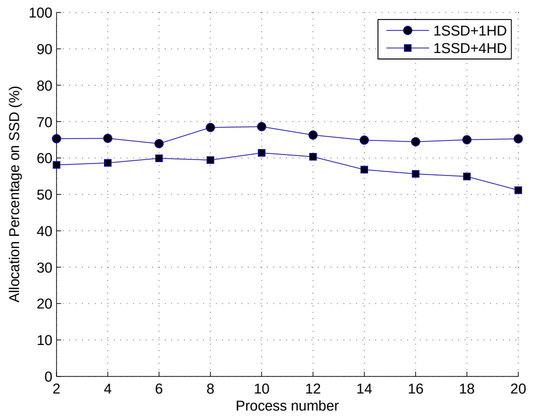


Fig. 13. Allocation percentage on SSD for the dbench workload in different configurations.

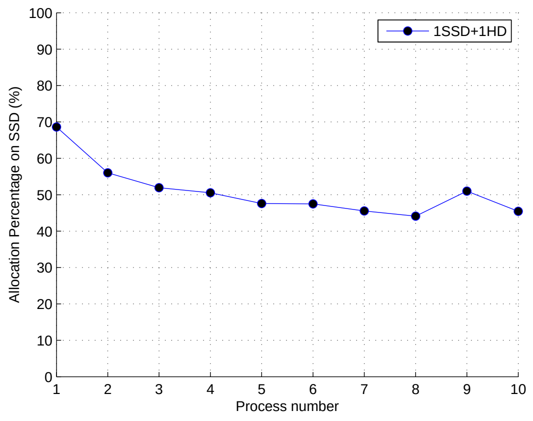


Fig. 14. Allocation percentage on SSD for the NetApp workload in 1SSD+1HD configuration.

TABLE III
PERFORMANCE IN A TRANSITIONING WORKLOAD.

	Throughput(Mbytes/sec)		Average Latency(ms)		Percentage on SSD	
	1 proc.	10 proc.	1 proc.	10 proc.	1 proc.	10 proc.
HD	12.3106	25.9799	0.200	0.894	0%	0%
SSD	12.413	23.6754	0.199	0.933	100%	100%
1SSD+1HD(our policy)	11.5769	36.8594	0.214	0.613	65.25%	55.08%

As an additional experiment, we ran an experiment where the workload initially consisted of one process of NetApp workload and halfway during the experiment, the workload transitioned into a load of 10 processes. The results of the experiment are shown in Table III. It is observed that our policy transitions the allocation to suit the changing workload and provides good performance across both the workloads.

As can be seen from the data, our policy adapts to different workloads and the configuration of the storage system to improve performance. The gains in performance are more significant with higher concurrency in the request stream of the workload.

IV. Related Work

Migration and caching are extensively studied in the context of memory and storage systems. Migration has been employed earlier in tape and disk storage systems [20] and in many file systems [14], [21] and in systems that employ dynamic data reorganization to improve performance [7], [8], [9], [15], [16], [17], [18], [22], [23], [39]. While some of these systems employed techniques to migrate data from faster devices to slower devices in order to ensure sufficient space is available on the faster devices for new allocations, migrated data in these systems tends to be “cold” data that hasn't been accessed for a while. Our system here can migrate both cold and hot data to the slower devices.

Migration has been employed to match data access characteristics to the device performance characteristics. For example, AUTORAID employed mirrored and RAID devices and migrated data to appropriate devices based on the read/write access patterns of blocks [3]. In our earlier work [32], we tried to match read/write requests to the observed read/write access characteristics of different device, SSDs and HDs, in a hybrid storage system, and employed migration as a primary vehicle for matching the block characteristics to device characteristics. In our current system described in this paper, we did not make that distinction and it is possible to combine the results from the earlier work to improve the performance further. We only consider aggregate request response time as a performance metric here.

Migration has also been employed for balancing load across networked storage systems [10] and in memory systems in multiprocessors, for example in [12]. Request migration has been employed to balance load in web based servers, for example in [13]. While we employ migration here for load balancing, we also used allocation as a vehicle for dynamically balancing the load across the different devices in our system.

Non-uniform striping strategies have been advocated before for utilizing disks of different capacity and performance [33], [34]. However, these strategies employed measurement characteristics that did not consider workloads or dynamic performance characteristics such as waiting times. Our approach here is guided by dynamically measured performance measures and hence can adapt to changing workloads and configurations without preset allocation ratios.

Several papers have recently looked at issues in organizing SSDs and magnetic disks in hybrid storage systems. A file system level approach has been used in [2]. In [36], magnetic disks are used as write cache to improve the performance as well as extend the lifetime of SSD devices. SSDs are employed to improve the system performance in [24], [25]. A special file system is designed for flash based devices in [35]. Performance issues of internal SSD organization and algorithms are well studied. For example, [30] presented how to build a high-performance SSD device. [26] focused on improving random writes for SSD devices. [27] proposed a new page allocation scheme for flash based file systems. [28] worked on garbage collection and [29] on wear-leveling for flash based storage systems. Characterization of SSD organization parameters also has been studied in [37], [38]. [31] provides a very good survey of the current state of algorithms and data structures designed for flash storages.

V. Conclusion

Our work considered block allocation and migration as a means for balancing the response times of devices across a workload. Dynamically observed performance of the devices is used to guide these decisions. In this paper, we only considered the aggregate performance of the devices irrespective of the nature of the requests (reads/writes, small/large etc.). Our policy allowed data to be allocated to slower devices before the space on the faster devices is exhausted. Our performance-driven allocation and migration improved performance of the hybrid storage system, both in improving latency of the requests and the throughput of the requests. We also showed that our policy adapted to different configurations of the storage system under different workloads.

In the future, we plan to consider the nature of the requests and the performance of the requests on different devices to improve performance further, in combination with the allocation based policy presented in this paper.

VI. acknowledgements

This work is supported in part by NSF grants 0702012 and 0621410. We thank Jiri Schindler of NetApp for providing traces. Part of this work was done while the second author was on a sabbatical at University of Carlos III and IMDEA NETWORKS in Madrid, Spain. We thank all reviewers for their thoughtful comments and suggestions to improve this paper.

References

[1] J. G. Wardrop. Some theoretical aspects of road traffic research. Proc. of Inst. of Civil Engineers, part II, vol.1, 1952.

[2] J. Garrison, A. L. N. Reddy. Umbrella File system: Storage management across heterogeneous devices. Texas A&M University Tech. Report, Decc. 2007.

[3] J. Wilkes, R. A. Golding, C. Staelin, T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Trans. Comput. Syst.* 14(1): 108-136 (1996).

[4] Dbench benchmark. Available from ftp://samba.org/pub/tridge/dbench/

[5] V. Borkar and P. R. Kumar. Dynamic Cesaro-Wardrop Equilibration in Networks. *IEEE Trans. on Automatic Control* 48(3): 382-396 (2003).

[6] F. Shu and N. Obr. Data set management commands proposal for ATA8-ACS2. www.t13.org, Dec. 2007.

[7] B. Gavish and O. Sheng. Dynamic File Migration in Distributed Computer Systems. *Commun. ACM*, pages: 177-189, 1990.

[8] E. Anderson et al., Hippodrome: Running circles around storage administration. *USENIX FAST Conf.*, Jan. 2002.

[9] L. Yin, S. Uttamchandani and R. Katz. SmartMig: Risk-modulated Proactive Data Migration for Maximizing Storage System Utility. In *Proc. of IEEE MSST* (2006).

[10] S. Kang and A. L. N. Reddy. User-centric data migration in networked storage devices. Proc. of IPDPS, Apr. 2007.

[11] A. W. Lueng, S. Pasupathy, G. Goodson and E. L. Miller. Measurement and Analysis of large-scale network file system workloads. Usenix Technical Conf., 2008.

[12] E. P. Markatos and T. J. LeBlanc. Load Balancing vs. Locality Management in Shared-Memory Multiprocessors. *Tech. Report: TR399, University of Rochester*, 1991.

[13] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proc. of ASPLOS* (1998).

[14] E. Miller and R. Katz. An Analysis of File Migration in a UNIX Supercomputing Environment. In *Proc. of USENIX* (1993).

[15] M. Lubeck, D. Geppert, K. Nienartowicz, M. Nowak and A. Valassi. An Overview of a Large-Scale Data Migration. In *Proc. of IEEE MSST* (2003).

[16] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *Proc. of USENIX Winter* (1992).

[17] C. Lu, G. A. Alvarez and J. Wilkes Aqueduct: online data migration with performance guarantees In *Proc. of FAST* (2002).

[18] R. Honicky and E. Miller. A Fast Algorithm for Online Placement and Reorganization of Replicated Data. In *Proc. of IPDPS* (2003).

[19] C. Wu and R. Burns. Handling Heterogeneity in Shared-Disk File Systems. In *Proc. of SC* (2003).

[20] A. J. Smith. Long Term File Migration: Development and Evaluation of Algorithms. *Communications of ACM* 24(8): 521-532 (1981).

[21] V. Cate and T. Gross. Combining the Concepts of Compression and Caching for a Two-level File System. In *Proc. of ASPLOS* (1991).

[22] R. Wang, T.Anderson and D. Patterson. Virtual log based file systems for a programmable disk. *Proc. of OSDI*, 1999.

[23] S. D. Carson and P. F. Reynolds, Adaptive disk reorganization. *Tech. Rep.: UMIACS-TR-89-4, University of Maryland, College Park, Maryland*, January 1989.

[24] I. Koltsidas and S. Viglas. Flashing up the storage layer. *Proc. of VLDB*, Aug. 2008.

[25] T. Kgil, D. Roberts and T. Mudge. Improving NAND Flash Based Disk Caches. *Proc. of ACM Int. Symp. Computer Architecture*, June 2008.

[26] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. *Proc. of USENIX FAST Conf.*, Feb. 2008.

[27] S. Baek et al. Uniformity improving page allocation for flash memory file systems. *Proc. of ACM EMSOFT*, Oct. 2007.

[28] J. Lee et al. Block recycling schemes and their cost-based optimization in NAND flash memory based storage system. *Proc. of ACM EMSOFT*, Oct. 2007.

[29] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. *Proc. of ACM SAC Conf.*, Mar. 2007.

[30] A. Birrell, M. Isard. C. Thacker and T. Wobber. A design for high-performance flash disks. *ACM SIGOPS Oper. Syst. Rev.*, 2007.

[31] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, June 2005.

[32] X. Wu and A. L. N. Reddy. Managing storage space in a flash and disk hybrid storage system. IEEE MASCOTS Conf., 2009.

[33] S. Kashyap and S. Khuller. Algorithms for non-uniform sized data placement on parallel disks. Foundations of software technology and theoretical computer science. Springerlink., 2003.

[34] J.Wolf, H.Shachnai and P. Yu. DASD Dancing: A disk load-balancing optimization scheme for on-demand video-on-demand computer systems. ACM SIGMETRICS, 1995.

[35] W. K. Josephson, L.A. bongo, D. Flynn and Kai Li. DFS: A new file system for virtualized flash storage. Proc. of USENIX FAST, 2010.

[36] G. Soundararajan, V. Prabhakaran, M. Balakrishnan and T. Wobber. Extending SSD lifetimes with disk-based write caches. Proc. of USENIX FAST, 2010.

[37] J-H. Kim, D. Jung, J-S. Kim and J. Huh. A methodology for extracting performance parameters in solid state disks (SSDs). Proc. of IEEE MASCOTS, 2009.

[38] F. Chen, D. Koufaty and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. Proc. of ACM SIGMETRICS, 2009.

[39] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for Self-Optimizing Storage Systems. Proc. of USENIX FAST, 2009.