SLO-aware Hybrid Store

Priya Sehgal, Kaladhar Voruganti, Rajesh Sundaram
NetApp Inc.
Email: {priya.sehgal, kaladhar.voruganti, rajesh.sundaram}@netapp.com

Abstract—**In the past storage vendors used different types of storage depending upon the type of workload. For example, they used Solid State Drives (SSDs) or FC hard disks (HDD) for online transaction, while SATA for archival type workloads. However, recently many storage vendors are designing hybrid SSD/HDD based systems that can satisfy multiple service level objectives (SLOs) of different workloads all placed together in one storage box, at better cost points. The combination is achieved by using SSDs as a read-write cache while HDD as a permanent store. In this paper we present an SLO based resource management algorithm that controls the amount of SSD given to a particular workload. This algorithm solves following problems: 1) it ensures that workloads do not interfere with each other 2) it ensure that we do not overprovision (cost wise) the amount of SSD allocated to a workload to satisfy its SLO (latency requirement) and 3) dynamically adjust SSD allocated in light of changing workload characteristics (i.e., provide only required amount of SSD). We have implemented our algorithm in a prototype Hybrid Store, and have tested its efficacy using many real workloads. Our algorithm satisfies latency SLOs almost always by utilizing close to optimal amount of SSD and saving 6-50% of SSD space compared to the naïve algorithm.**

Keywords-*SSD;Flash; SLO; Hybrid Storage; dynamic partition*

I. INTRODUCTION

A cloud is usually characterized by multiple applications/workloads that share the same set of storage system. Each of these applications have certain set of Service Level Objectives (SLOs) [1], namely performance (average I/O latency, average throughput), capacity, reliability, and security (retention of data, encryption), etc. One of the aims of a cloud provider is to satisfy all the SLOs at the lowest cost. With the emergence of Flash and SSDs, storage vendors are combining it with HDDs to satisfy SLOs at better cost points than pure Flash/SSD or pure HDD based solutions. They are placing SSDs as a read-write cache in front of HDDs or using SSD as a different storage tier. We call such a solution a *Hybrid Storage* (HyS). Examples include NetApp® FlashCache [2], EMC²® FAST [3], etc.

One problem common to all the above mentioned multi-tenant Hybrid Storage systems is the lack of *SLO based management for SSDs or flash*. SSD cache is shared across the different workloads i.e., there is a common LRU queue to manage the SSD cache space across all the workloads. Thus, Hybrid Store treats all the workloads in similar fashion. This leads to three major problems: (1) SLO inversion of workloads, (2) SLO violation of few workloads, and (3) suboptimal SSD utilization.

With a single shared LRU queue in the SSD tier of Hybrid Store, workloads that do not have a very stringent latency requirement can utilize more than the required amount of SSD to meet their target latency, depriving the other stricter SLO workloads of SSD resources and hurting their performance. This results in SLO inversion, e.g., Silver customer perceiving Gold customer experience and vice-versa. Further, this results in SLO violation of workloads that have stringent latency requirements. Since the SSD resource is not utilized by the right set of workloads, it leads to sub-optimal SSD utilization.

This paper brings the concept of SLO to the SSD caching layer in Hybrid Store. In this paper we focus only on performance SLO and more specifically *latency* SLO. We have designed and implemented a light-weight feedback-based proportional controller, called *Error-aware Feedback Controller* (EAFC) that dynamically sizes the SSD caching tier of Hybrid Store for each workload, depending upon its latency requirements and workload characteristics. Similar idea can be applied on any other shared resource in the storage stack, but we have limited our study to only SSD caching layer when used as a *read cache* only.

The major contribution of this work is design and implementation of per-workload feedback controller that sizes the SSD partition on a HyS for each workload close to optimal so as to meet its respective latency requirement, and dynamically adapt the partition size to changes in workload and working set sizes. The key insights gained by this work on SSD cache sizing are:

- In order to meet latency requirement only a fraction of working set size needs to be cached depending upon workload characteristics and working set size.
- SSD cache size should keep some headroom to accommodate workload changes: if the sizing algorithm tries to meet 75th percentile observed latency, average target latency is always met.
- We do not require a lot of history to set the partition size appropriately. A simple feedback controller algorithm looking at only a few 100 history points works accurately.

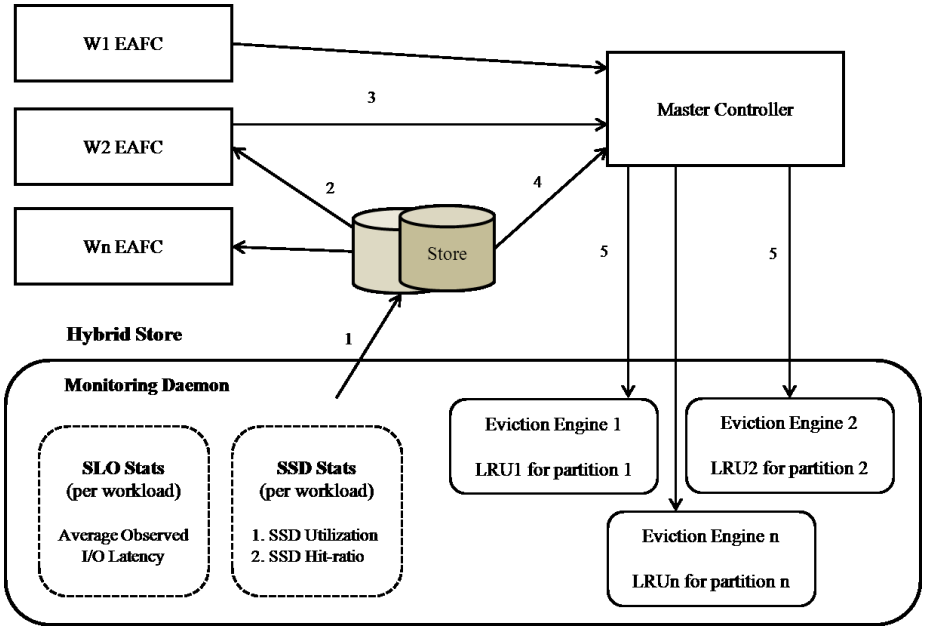

Figure 1. High Level Architecture of the Sizing Controller

II. Related Work

In the past, there has been a lot of research that focused on resource management to achieve service differentiation (QoS), and fairness for multiple resources. Static and dynamic partitioning has been looked upon as one of the approaches to achieve these objectives [5-9]. CacheCOW [11] presents algorithms to dynamically allocate cache space among multiple classes of workloads to meet QoS differentiation and performance maximization. Lu, et al. [10] proposes a QoS-control paradigm that is based on adaptive control theory. Guerra et al. [12] implements a dynamic tiering solution that dynamically places extents across SSDs and HDDs to satisfy performance requirement while minimizing dynamic power consumption.

In our work we present an adaptable, feedback heuristic controller that dynamically changes SSD cache sizes allocated to workloads depending upon their expected and observed latencies and SSD usage. It is related to work presented in [10] and [11], but our sizing decisions are made keeping the SSD media idiosyncrasies, like SSD cache warm up time, in mind.

III. High Level Architecture

Service level objective (SLO) is a technology independent term for specifying the services desired from a data set [1]. For example, performance level is described primarily in terms of a target latency and target throughput. Performance SLO, say latency, can be expressed as service level by a tuple like `<max latency, conformance window, conformance percentage>`. Here, latency SLO is considered to be met if average observed latency over a period of conformance window is well below max latency for at-least conformance percentage times. This is the definition of SLO at a very high level in storage stack. But, at lower layers like file system (where SSD caching logic resides), the SLO terminology changes; the file system (FS) layer sees stringent deadlines associated with each I/O packet, after deducting all the queuing delays at the higher layers. FS layer cannot borrow the same notion of conformance window and percentage, but should meet the deadlines as much as possible. It is also not necessary for FS to meet per-I/O deadline 100% of time, since end user will only perceive average performance and it would result in caching the entire working set in SSD cache, thereby using SSD sub-optimally. Thus, it is essential that caching layer of FS in HyS tries to meet average latency over a period of time called as controller window (discussed later).

Hybrid Store consists of a set of HDDs used for permanent storage that are front-ended with SSDs used for read caching. HDDs are configured with RAID and provision multiple data sets or logical data volumes. All the I/O streams coming to one volume are termed as a *workload*. In this work, we assume that there is only one user application, say Oracle DB, associated with one workload or volume. We assume that the storage system provides interface to set latency SLO, as described above, on a per-workload basis. As discussed later in this section, we assume HyS to provide an infrastructure that allows gathering and reporting of important storage and SLO statistics for every workload at the caching layer (FS layer). Lastly, we consider that HyS supports dynamically partitioned SSD cache – one LRU per workload, where LRU size can be changed.

To implement a dynamic SSD cache sizing for every workload depending upon the latency requirement, we designed a two-level controller: *per-workload feedback controller (or EAFC)* and a *master controller*, both depending on storage and SLO statistics reported by

monitoring daemon, as shown in Figure 1. The *eviction engine* is per workload. It is a daemon in HyS that aids in dynamic SSD partitioning of its respective workload: it is responsible for knowing the partition size set by EAFC for its workload and managing LRU. The eviction engine is woken up whenever the SSD utilization reaches a threshold, such that it can make place for new to-be inserted blocks in advance. The following paragraphs explain each of these modules and their interaction in more detail.

Monitoring Daemon: Assuming the storage system maintains statistics; this module collects these stats every 5 seconds, and stores it in a comma separated log file (Store in Figure 1). The per-workload statistics that are collected by the daemon include SLO stats like average observed I/O latency at the HyS layer and SSD statistics like SSD cache used and SSD cache hit ratio.

Per-workload Feedback Controller: This is the error-aware feedback controller, one per workload, shown in Figure 2. If we have 10 workloads each tied to different volumes on a HyS; we will have 10 different EAFC. EAFC sleeps for most of the time and wakes up after every *n* seconds - called the controller window (cwnd). When EAFC wakes up, it looks up the statistics collected by monitoring daemon in the last cwnd and determines the partition size for its workload (explained in Section IV). As the feedback controller works on the most recent history of statistics, it is adaptive to workload and working set size changes.

Master Controller: The master controller is an arbitrator of all the EAFCs. In contrast to EAFC, the master controller has a global view of all the workloads i.e., their statistics and priority. In case of SSD space contention, the master controller solves a variation of *knapsack problem* such that most of the high priority workloads meet their SLOs at the cost of violating SLOs of few low priority ones. The master controller chooses low priority victim workloads whose SSD sizes are shrunk to benefit higher priority workloads.

In this paper, we have designed and implemented monitoring daemon and EAFC. Here, the EAFC directly talks to its respective eviction engine to modify the partition size. We have not implemented the master controller and plan to do it in the future.

IV. FEEDBACK CONTROLLER

A. Design Dimensions

While designing the per-workload EAFC controller, we considered various dimensions:

Un-partitioned vs. Partitioned Cache: In a multi-workload environment, it is difficult to avoid interference between competing workloads in an un-partitioned cache. One of the most prevalent methods to achieve differentiated QoS is through resource partitioning [5-11]. Thus, we chose to implement a per-workload SSD partition cache, where each eviction engine manages block insertion/eviction from its own private LRU queue (partition).

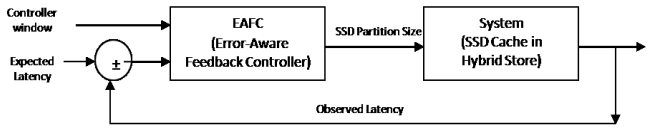

Figure 2. Error-Aware Feedback Controller

Static vs. Dynamic Cache Partitioning: Partitioning can primarily be implemented in two ways: static or dynamic. Static partitioning could set the partition size for a particular workload to a fixed value depending upon it working set size, if known a-priori. Problems with static cache sizing are that it is not adaptive to latency target, working set size and workload characteristics changes, leading to over-provisioning or under-provisioning the workload. Hence, we chose dynamic cache sizing technique.

Aggressive vs. Conservative SSD Partition Decrease: The feedback controller increases the partition size when the SLO is not met and decreases it when observed SLO is better than expected. We found experimentally that if the partition size is decreased as soon as EAFC sees minimal improvement in SLO, it leads to EAFC oscillation and sometimes SLO violation. This is because of unnecessary evictions of some useful blocks. Hence, we need a conservative decrease policy.

We ran a few experiments with static cache size and observed that even if the partition size was kept fixed, the observed latency oscillates because of the very nature of the workload and SSD cache misses. On analysis of the results, we found that for every static size, less than the working set size, the observed latency oscillated within a range – *min-max range*. Thus, if for every expected latency EAFC tries to operate within a min-max range, where max latency is same as expected latency, while min latency is some fraction of max latency, SLO violation can be mitigated. In the min-max range operation, EAFC decreases the partition size only if observed latency goes below min latency. We observed that with min-max range of operation, the oscillations reduced as the partition size remained constant for a substantial amount of time, thereby avoiding unnecessary evictions. In all our experiments min latency was set to half the target latency.

B. Feedback Controller Algorithm

The EAFC is a simple feedback based *proportional controller*, which performs a proportional increase if the SLO is not met and a proportional decrease if the SLO is beyond expectation i.e., below minimum latency, otherwise it keeps the partition size unchanged (shown in Figure 2.). The controller decides the next steps depending on the statistics collected in last cwnd only.

Figure 3. provides the pseudo-code of EAFC. The feedback controller receives two types of inputs: external and feedback inputs. Feedback input is the output from the system (i.e., HyS), which is collected by the monitoring daemon, while external input is provided by the user, e.g., controller window *cwnd*. Note that EAFC compares the *75th percentile* of observed latency ($latency_{obs}$) with the expected latency ($latency_{max}$) so that the average observed latency

meets the SLO almost all the time. This enables EAFC to keep enough extra SSD space so as to avoid SLO violation in the average case. Further, experimental results showed that operating at 75th percentile is better than stringent latency (e.g., 90th, 99th percentile), as the latter resulted is partition size oscillation and average latency violation.

The algorithm is broken into 3 cases. In Case 1, $latency_{obs}$ is within min-max range, hence the controller does nothing and goes to sleep. Case 2 handles SLO violation; if the SSD utilization is below T_{util}, workload has not used up the available SSD and hence EAFC does nothing. If the SSD utilization is above T_{util}, $cache\_sz_{cur}$ is increased by a proportion (K_{pinc}) of the error in $latency_{obs}$ and $latency_{max}$. In Case 3, when SLO exceeds expectation and $latency_{obs}$ is below $latency_{min}$, EAFC calculates the error in latencies and decreases the SSD cache size by proportion (K_{pdec}) of this error.

V. EVALUATION

We used FileBench[4] Web server and SPECsfs2008[13] like workload to study the effectiveness of the feedback sizing controller. We evaluated the efficacy of EAFC in terms of meeting target latency with close to optimal amount of SSD under following use cases: (a) Changing the expected latency for workload, (b) Varying working set sizes of the same workload, and (c) Varying the workload intensity or load.

Our experimental setup consisted of a prototype Hybrid Store with 1TB of HDD space exposed as one volume, 160 GB of SSD space used as *read cache* for this volume, and 16 GB of RAM. The 1TB volume was exported through NFS. All the I/O streams coming to this volume were denoted as a workload, which was tied up to a latency target. The server setup remained constant for all variations of the tests. The client setup was specific to individual experiment (discussed later).

Note that in all the graphs (Figure 4.) the left axis denotes average observed latency in milliseconds and right axis represents SSD size in GBs. The x-axis represents time in units of 10 minutes i.e., cwnd. In the graphs, *partition_sz* plot represents the SSD partition size set by the controller, while *Used_SSD* denotes the amount of SSD utilized by the workload in GB. As the eviction engine makes space for newly inserted blocks, the utilization does not go beyond 90% of partition size.

A. FileBench Web Server

For FileBench[4] web server workload, we used the same server setup as described earlier in this section. The NFS client setup consisted of two Linux (Ubuntu) machines each with 4 GB RAM and 2 CPUs. Each client mounted the NFS exported volume and created its dataset in a different directory in the mount point. We tried multiple variants of web server workload, but due to space constraints discuss the results of only two scenarios. The first test was to determine that the feedback controller adapts the SSD partition to the latency requirement of a workload. For this test, the web server workload consisted of a total of 400 threads (200 on each client) accessing a total of 1.6 million small files (~20KB), resulting in working set size of 40GB.

Notations:
cwnd: Controller (EAFC) window (10 minutes)
$latency_{obs}$: Observed latency (75th percentile) in last cwnd
$latency_{max}$: Average maximum expected latency at caching layer
$latency_{min}$: Average minimum expected latency at caching layer
ssd_util_{avg}: Average amount of SSD used
$cache\_sz_{cur}$: Current SSD partition size
$cache\_sz_{prev}$: SSD partition size in the last cwnd
K_{pinc}, K_{pdec}: Increase/decrease proportionality constants.
T_{inc}, T_{dec}: Threshold for multiplicative increase and decrease
T_{util}: SSD utilization threshold
External Inputs: cwnd, $latency_{max}$, K_{pinc}, K_{pdec} T_{inc}, T_{dec} T_{util}
Feedback Inputs: $latency_{obs}$, ssd_util_{avg}, $cache\_sz_{prev}$
Outputs: $cache\_sz_{cur}$
Initialization: $latency_{min} = 0.5 \times latency_{max}$

CASE 1: SLO met, but within min-max range
i.e., $latency_{max} \geq latency_{obs} \geq latency_{min}$
EAFC goes to sleep for cwnd without changing SSD partition size

CASE 2: SLO not met i.e., $latency_{obs} > latency_{max}$
if $(ssd\_util_{avg} \div cache\_sz_{prev}) \geq T_{util}$
$error = latency_{obs} - latency_{max}$
$ratio_{inc} = error \times K_{pinc}$.
if $(ratio_{inc} > T_{inc})$: $ratio_{inc} = T_{inc}$
$cache\_sz_{cur} = cache\_sz_{prev} (1 + ratio_{inc})$
else
$cache\_sz_{cur} = cache\_sz_{prev}$
CASE 3: SLO exceedingly met i.e., $latency_{obs} < latency_{min}$
$error = latency_{min} - latency_{obs}$
$ratio_{dec} = error \times K_{pdec}$
if $(ratio_{dec} > T_{dec})$: $ratio_{dec} = T_{dec}$
$cache\_sz_{cur} = cache\_sz_{prev} (1 - ratio_{dec})$

Figure 3. EAFC pseudo-code

Figure 4a and 4b show results of the first test for average expected latency requirements of 10ms and 5ms, respectively. If we compare the two figures, we find that for both 10ms and 5ms target requirement, the feedback controller is able to meet the average latency almost all the time. During the initial cache warm up time the observed latencies are higher than expected, but later the latencies remain well below the maximum expected latency target. For 10ms latency requirement, the average SSD partition size required was 39GB, while for 5ms it increased to 41 GB. Thus, the SSD partition size depends upon the expected latency requirement. When this test was executed on a no-controller setup (vanilla), SSD space utilized was 40GB and resulted in approximately 1ms observed latency. Considering the SSD utilization cannot go beyond 90% of partition size, the partition size required by vanilla case would be 44GB. If we compare the partition size required by vanilla case and EAFC, we see that the latter saves 6% and 11% SSD space for 5ms and 10ms, respectively, yet achieving 100% SLO conformance.

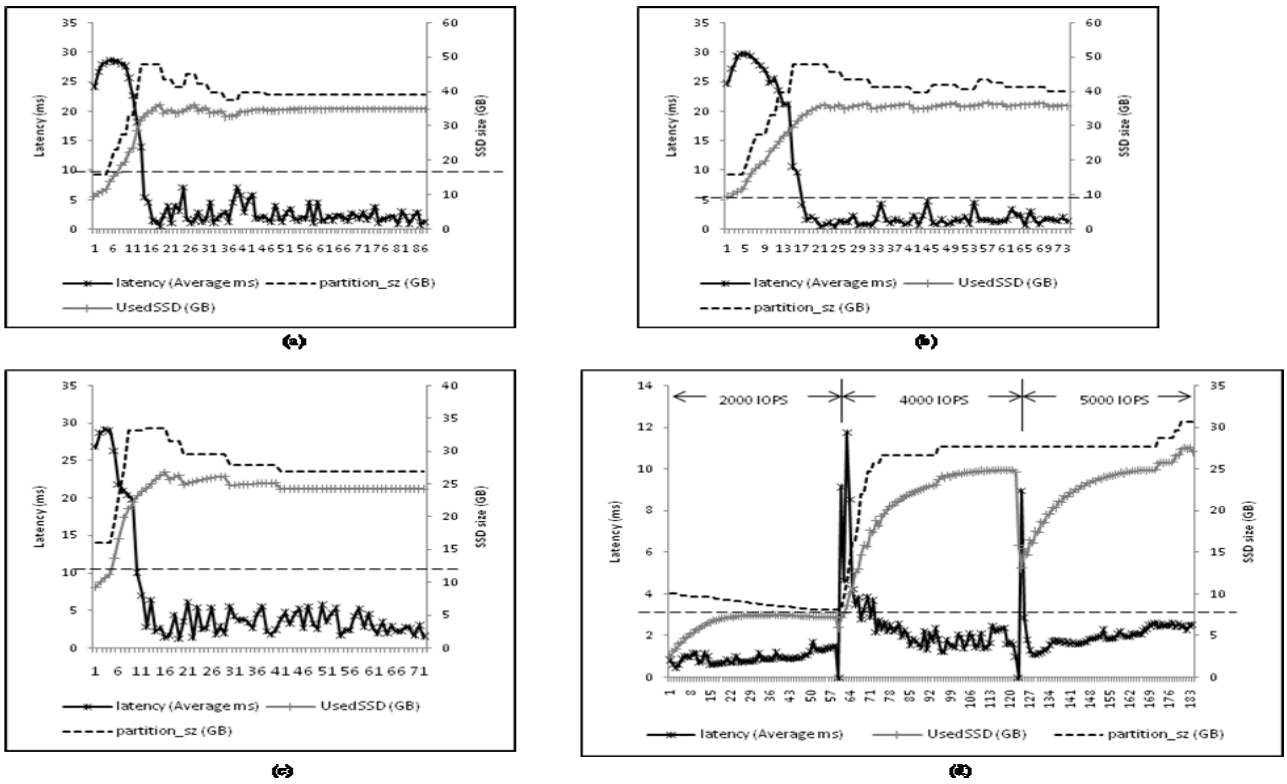

Figure 4. Average observed latency, SSD partition size and SSD utilization plots for FileBench and SPECsfs2008

In the second test, we tried to determine if the controller can adapt to changes in working set sizes. We repeated the web server workload with same client configuration, but for 30GB working set size (i.e., number of files per client was 750,000) and setting the expected latency to 10ms. Figure 4c shows the result of this experiment. If we compare Figure 4a and 4c, we see that our controller adapts to different working set sizes– average SSD partition size drops from 39GB to 28GB when the working set size decreases for SLO requirement of 10ms.

B. SPECsfs2008

We ran SPECsfs2008[13] like workload consisting of 20 threads, and increasing the total IOPS after every 5 hours; total IOPS being 2000, 4000, and 5000. This resulted in increase in both the workload intensity and working set. Figure 4d shows results of this 15 hour run with EAFC controlling the SSD size for SLO requirement of 3ms. The initial SSD partition was set to 10GB. We see that during the first 5 hours (when the IOPS = 2000), SLO is always met, hence the SSD partition size is decreased. This was because of high buffer cache hit ratio. As the load increases to 4000 and 5000 IOPS, the buffer cache miss increases and hence partition size rises in order to meet the SLO. Note that we see sudden rise in the observed latency at two points – these are the places where the workload's (SPECsfs) intensity increased, resulting in SSD cache misses. From the graph it is clear that our feedback controller sizes the cache close to optimal to meet the SLO – it adapted the size based on demand and tried to meet SLO almost all the time. We also found that EAFC saved around *50%* space compared to vanilla scenario.

VI. CONCLUSION

This paper presents the design, implementation and evaluation of error-aware feedback controller that sizes the SSD partition for a workload based on its latency requirement, thereby utilizing SSD efficiently yet meeting SLOs. Our results show that EAFC adapts to changes in workload intensity, working set size and latency requirements. In the future, we plan to implement the master controller that would arbitrate between multiple EAFC in case of SSD contention.

ACKNOWLEDGEMENT

We would like to acknowledge various NetApp ATG members who helped improve the work through many brainstorming sessions and reviewing the paper. We would also like to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] L. N. Bairavasundaram, G. Soundararajan, V. Mathur, K. Voruganti, and S. Kleiman, "Italian for Beginners: The Next Steps for SLO-Based Management", USENIX HotStorage '11.

[2] NetApp Flash Cache: http://www.netapp.com/us/products/ storage-systems/flash-cache/

[3] EMC2® FAST: http://www.emc.com/collateral/software/white-papers/h8046-clariion-celerra-unified-fast-cache-wp.pdf

[4] FileBench: http://sourceforge.net/projects/filebench/

[5] S.Kim, D. Chandra, and Y.Solihin, ”Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture”, In Proc. PACT’04.

[6] R.Lyer, “QoS: A framework for enabling qos in shared caches cmp platforms”, In Proc.ICS’04, pp.257-266, 2004.

[7] H. S. Stone, J. Turek, and J.L. Wolf, “Optimal partitioning of cache memory”, IEEE Transactions on Computers., 41(9), 1992.

[8] G. E. Suh, L. Rudolph, and S. Devadas, “Dynamic partitioning of shared cache memory”, In Journal of Supercomputing, 2004.

[9] M. K. Qureshi and Y.N. Patt, “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches”, In Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture.

[10] Y. Lu, T. Abdelzaher , C. Lu , and G. Tao, “An Adaptive Control Framework for QoS Guarantees and its Application to Differentiated Caching Services”, In Proc. Tenth International Workshop on Quality of Service, 2002.

[11] P. Goyal, D. Jadav, D. Modha, and R. Tewari, “CacheCOW: QoS for Storage System Cache”, In Eleventh International Workshop on Quality of Service, 2003.

[12] J. Guerra, H. Pucha, J. Glider, W. Belluomini and R. Rangaswami, “Cost Effective Storage using Extent Based Dynamic Tiering”, FAST 2011.

[13] SPECsfs2008: http://www.spec.org/sfs2008/