Improving I/O Performance Using Soft-QoS-Based Dynamic Storage Cache Partitioning

Christina M. Patrick
Penn State University
patrick@cse.psu.edu

Rajat Garg
Penn State University
rgarg@cse.psu.edu

Seung Woo Son
Argonne National Laboratory
sson@mcs.anl.gov

Mahmut Kandemir
Penn State University
kandemir@cse.psu.edu

***Abstract*—Resources are often shared to improve resource utilization and reduce costs. However, not all resources exhibit good performance when shared among multiple applications. The work presented here focuses on effectively managing a shared storage cache. To provide differentiated services to applications exercising a storage cache, we propose a novel scheme that uses curve fitting to dynamically partition the storage cache. Our scheme quickly adapts to application execution, showing increasing accuracy over time. It satisfies application QoS if it is possible to do so, maximizes the individual hit rates of the applications utilizing the cache, and consequently increases the overall storage cache hit rate. Through extensive trace-driven simulation, we show that our storage cache partitioning strategy not only effectively insulates multiple applications from one another but also provides QoS guarantees to applications over a long period of execution time. Using our partitioning strategy, we were able to increase the individual storage cache hit rates of the applications by 67% and 53% over the no-partitioning and equal-partitioning schemes, respectively. Additionally, we improved the overall cache hit rates of the entire storage system by 11% and 12.9% over the no-partitioning and equal-partitioning schemes, respectively, while meeting the QoS goals all the time.**

I. Introduction

Resources are often shared to help reduce administration and maintenance costs, avoid underutilization of resources, and help the bursty workloads utilize resources that would otherwise be left idle. However, not all resources exhibit good performance when shared among multiple applications. Prior research has shown that sharing the same storage cache among multiple, simultaneously executing applications can degrade cache performance significantly and lead to unpredictable performance at the user end [1].

The work presented in this paper focuses on improving storage caching. We refer to the kernel buffer cache that resides in main memory as the "storage cache." Extensive research [2], [3], [4], [5], [6], [7], [8], [9] has been done on improving the effectiveness of storage caching. However, little research has been done on providing quality of service (QoS) guarantees to multiple applications that exercise storage caches.

Storage cache partitioning has been proposed as a solution to providing QoS guarantees to applications sharing a storage cache. A storage cache can be partitioned statically or dynamically. The easiest strategy is to partition the storage cache *equally* among competing applications. While this scheme is easy to implement and enables application isolation, it, however, leads to underutilization of resources. Moreover, a cache-hungry application can suffer badly if its allocated cache space is not sufficient to hold its entire working set. Furthermore, in general, any static partitioning has no way of adapting to the dynamic modulations in cache space requirements. Consequently, dynamic cache partitioning seems to be a promising alternative to static partitioning.

However, optimal dynamic partitioning of storage cache is not trivial in practice because (i) applications typically have independent QoS demands that may not be possible to satisfy at the same time; (ii) effects of cache space allocation are not visible immediately but accrue over time; (iii) cache space allocation of one application significantly affects the cache hit rates of other applications in the cache; (iv) the effect of cache space allocation depends on application data reuse and locality as well as data access pattern, and more cache allocation does not necessarily imply better hit rates [10]; and (v) the hit rate of an application depends on the phase in which the application is executing, making it difficult to implement a good dynamic scheme. Hence, techniques that are used to enforce QoS in resources such as CPU and network bandwidth cannot be applied easily to cache space allocation. Motivated by these observations, this paper makes the following contributions:

- We propose a QoS-aware dynamic storage cache partitioning scheme that employs curve fitting [11] to dynamically partition the cache space among competing applications. Our scheme could be used by a service provider to consolidate several applications onto a single system in order to decrease costs, at the same time maintaining a performance equivalent to a stand-alone system where every application has its own dedicated cache. The proposed scheme uses history information to predict future cache space requirements. Since it employs curve fitting, this scheme improves over time and is able to capture the dynamic behavior of applications. The results of our cache partitioning scheme are twofold. First, our algorithm adapts the cache partition sizes among competing applications in order to satisfy each application's QoS. Second, we distribute the remaining cache space among the competing applications in a manner that helps an application whose QoS is satisfiable to achieve its maximal (cache) hit rate possible. Improving the storage cache hit rate of an application is important because the storage cache hit rate directly translates to execution time [1], [12], [13], [6], [14], [15], [7], [16] of an I/O-intensive application. Since we partition the remaining cache capacity carefully across applications, this also helps

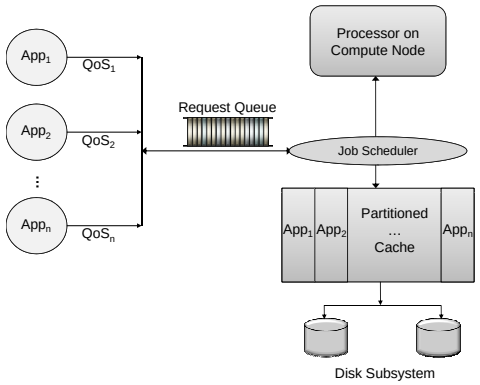


Fig. 1. Shared storage cache architecture considered in our work.

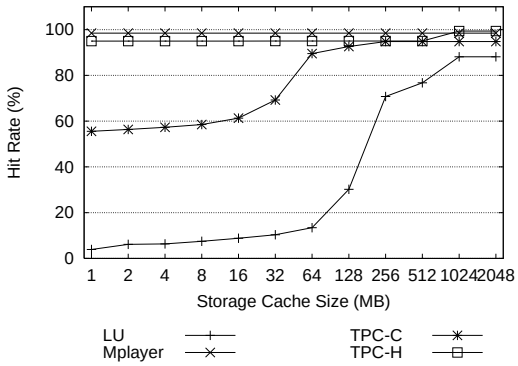


Fig. 2. Individual hit rates of different applications for varying storage cache sizes. Applications used above have been described in Table III. Details of setup used to collect this data are given in Section V-A. Note that prior work [17] reports a similar behavior for tpc-h.

maximize the overall storage cache hit rate. Our partitioning scheme has a low overhead and focuses on soft-QoS rather than hard-QoS class of applications.

• Using extensive trace-driven simulation, we first demonstrate the interference caused by the sharing of the storage cache among multiple applications. Our experiments show that the proposed dynamic storage cache partitioning scheme is able to achieve the specified hit rate (QoS) if possible and generates better results than both the equal-partitioning scheme and no-partitioning scheme. For instance, when we used a mixed workload of tpc-h, tpc-c, mplayer, and lu, we were able to increase the hit rate of lu by 67% in a 1 GB cache over the no-partitioning scheme. Additionally, when we used a workload with multiple instances of lu, we were able to increase the overall hit rate of the storage cache by 11% in a 256 MB cache compared to the no-partitioning scheme.

Section II describes the shared storage cache architecture we target. The details of our storage cache partitioning scheme are given in Section III. Section IV discusses the two base schemes against which we compare our proposed partitioning approach. Section V describes our experimental setup, the workloads we use, and the results obtained using our storage cache partitioning algorithm. Section VI discusses prior research on storage cache management, and Section VII summarizes our work.

II. Target Architecture

The work we present in this paper can be used to improve the performance of a consolidated cache in an SMP/CMP system, a consolidated NAS/SAN server that serves multiple clients. We tested our scheme on the target shared storage cache architecture shown in Fig 1. All applications running on a compute node have access to a common shared cache.[1] The specific storage cache architecture simulated in this work is similar to the buffer cache in Linux. The kernel buffer cache that resides in main memory acts as a storage cache[2] in Linux. In addition to a global free list, Linux maintains a list of buffers for disks. In order to facilitate fast search, the Linux kernel implements a hash table. When there is a request for data, the disk buffer list is searched. If it is already in the buffer cache, the pointers are rearranged to reflect the access (depending on the cache replacement policy). If not, a buffer from the free list is allocated for the data. If no buffers are on the free list, the data in one of the buffers in use is replaced with the new requested data (the victim chosen depends on the replacement policy). In our experiments, we study this storage cache behavior in detail. We use the Linux default cache replacement policy, LRU, for all our experiments.

III. Our Approach

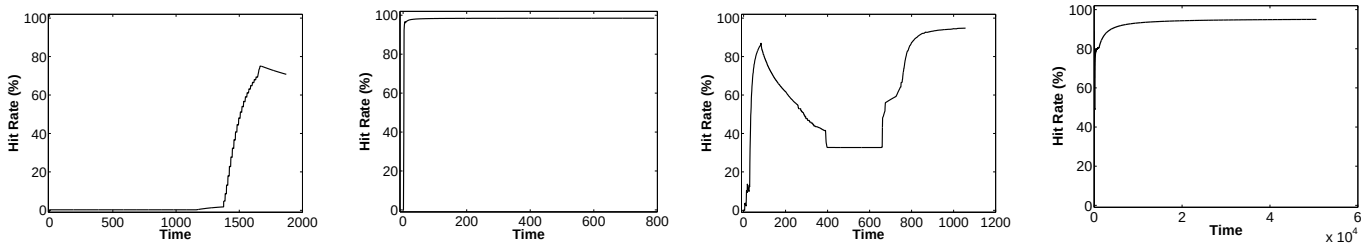
We start by presenting some experimental data that illustrates the need for dynamically changing storage cache partitioning at runtime. Fig 2 shows the cache hit rates of four different applications under varying storage cache sizes when each of them solely occupies the cache. One can observe from this figure that different applications behave differently as the storage cache capacity is increased. For example, we see that the hit rates of lu and tpc-c are increasing with increase in the storage cache size (up to a certain point beyond which the curves become flat), whereas the hit rates of mplayer and tpc-h are more or less constant irrespective of the cache size. Hence, we may conclude that some applications benefit considerably when they are given more storage cache space, while others do not necessarily benefit substantially when given increasing cache size. Clearly, a good dynamic partitioning policy should be able to recognize the difference between these two categories of applications and allocate storage cache partitions accordingly.

Fig 3 plots the hit rate behavior of the same applications *over time* given a constant cache size.[3] As can be seen from this figure, the hit rate of the application is subject to the phase in which the application is executing and varies from

[1]The compute node may also act as a server serving multiple clients. However, we do not study the interaction of multilevel caches in this paper.

[2]We distinguish the storage cache from the cache that is physically present on the disk, which we will refer to as the "disk cache."

[3]Note that the sampling time (10 ms) is much smaller than what it would have been in a real execution, since we are using a simulator that is orders of magnitude faster than a real execution environment. The simulator does not actually access the disk. It calculates only the time incurred by a hit/miss. We also plotted the graphs using different sampling intervals ranging from 10 ms to several seconds. The shape of the graphs do not change.



(a) LU (b) Mplayer (c) TPC-C (d) TPC-H query 17

Fig. 3. Hit rates of different applications over time in a 256 MB storage cache. Hit rate sampling interval is 10 ms. Applications used above are described in Table III. Details of setup used to collect this data are given in Section V-A.

phase to phase. Therefore, we can appreciate the difficulties in implementing a good dynamic partitioning scheme that must take into account this varying application behavior. The scheme we propose and evaluate in the rest of this paper addresses these dynamic variations of cache hit rates.

A. QoS Specification and Optimization Goal

The goal of our scheme is to ensure that the access latency of an application i must be less than or equal to its specified access latency (QoS), T_{QoS_i} over a time period T_e, where T_e may be on the order of minutes to a few hours. If t_{e_i} is the execution time of application i, t_c is the time required to access the cache, t_d is the average disk access time, and h_i is the application hit rate, then its average execution time can be estimated as $t_{e_i} = t_c \times h_i + t_d \times (1 - h_i)$. In practice, $t_c \ll t_d$. For instance, Wong and Wilkes [7] use $t_c = 0.2$ ms and $t_d = 10$ ms in their modeling.

Thus, to reduce application execution time, we try to increase the hit rate of the application. We define the "overall cache hit rate" as the ratio of the total number of storage cache hits to the total number of storage cache accesses made by all concurrently running applications since the time the first application was instantiated on the system (i.e., since time = 0). The overall storage cache hit rate is different from the individual hit rate of an application (which accounts for the hits and accesses of the individual application only since its instantiation on the system) and captures all hits and accesses coming from all applications that exercise the storage system since the initiation of the first application on the system.

To decrease the overall execution time, we must increase the overall storage cache hit rate. To do so, we try to effectively increase the individual application hit rates while maintaining the application QoS. Our scheme satisfies the application QoS if it is possible to do so and decreases the overall execution time by controlling the application hit rates. We do so by controlling the cache space allocated on a per application basis. Thus, for application i, if T_{QoS_i} is the QoS latency specified and H_{QoS_i} is the application hit rate corresponding to this latency, then our scheme works such that $t_{e_i} \leq T_{QoS_i} \Rightarrow h_i \geq H_{QoS_i}$

Our proposed idea could be employed especially by service providers who want to use consolidated (instead of dedicated)

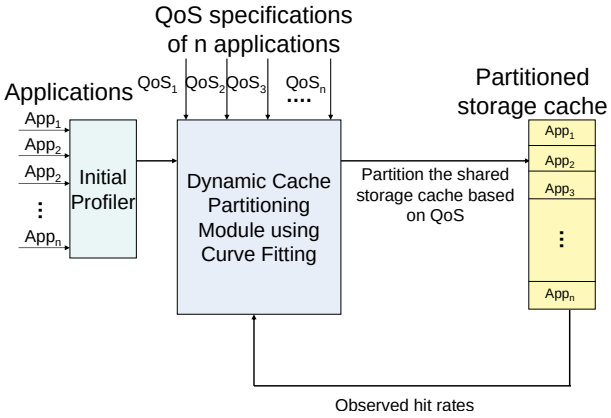


Fig. 4. High-level operation of the storage cache partitioning scheme.

infrastructure to reduce costs and provide isolation while maintaining performance equivalent to a stand-alone system. Our scheme is soft-QoS- rather than hard-QoS-centric. That is, we do not restrict applications from entering a system (oversubscription); rather we try to satisfy the specified QoS of an application as much as possible (best effort). If desired, however, our policies could be easily adapted to employ throttling.

B. QoS-Based Shared Cache Partitioning

Fig 4 shows our high-level architecture. The pseudo-code for our dynamic cache partitioning algorithm is given in Fig 5. It is divided into several steps that are explained below in detail.

Recall that our first goal is to satisfy the QoS specified for an application. To do so, we need to determine its cache space allocation. Specifically, we need to predict the performance (hit rate) of the application for a cache capacity range (so that we can select the right one). This prediction is done by using *curve fitting* and is illustrated in Fig 6 for the tpc-c case.

Our curve fitting uses piecewise linear interpolation to construct a curve from the available data.[4] However, note that a linear model in this context does not necessarily imply a

[4]Selection of the model (linear, spline, etc.) used for curve fitting is orthogonal to the focus of this paper. However, we select linear curve fitting as it has a low computational overhead and is fast compared to other methods of curve fitting.

C_{tot}: total cache size
C_{free}: total free cache size
N: number of apps running concurrently
QoS_j: QoS of app j
Hit_j: hits of app j
$Accesses_j$: accesses of app j
C_j: cache partition size of app j
F_j: flagged status of app j
$numflag$: number of apps whose QoS is not satisfied

Partition()
// Profiling
for $j \leftarrow 0..N-1$ **do**
$HR_j \leftarrow \frac{Hit_j}{Accesses_j} \times 100$

// Repartitioning step
$C_{free} \leftarrow C_{tot}$
$sum \leftarrow 0$
for $j \leftarrow 0..N-1$ **do**
$C_j \leftarrow curve\_fitting(QoS_j)$
$delta_j \leftarrow max(HR_j) - HR_j$
$sum \leftarrow sum + delta_j$

$numflag \leftarrow 0$

// Flagging the apps
for $j \leftarrow 0..N-1$ **do**
if $C_{free} \geq C_j$
$C_{free} \leftarrow C_{free} - C_j$
$F_j \leftarrow 0$
else
$C_j \leftarrow 0$
$F_j \leftarrow 1$
$numflag \leftarrow numflag + 1$

// Non-flagged apps
if $numflag = 0$
for $j \leftarrow 0..N-1$ **do**
$C_j \leftarrow C_j + C_{free} \times \frac{delta_j}{sum}$

// Flagged apps
if $numflag \neq 0$
for $j \leftarrow 0..N-1$ **do**
if $F_j = 1$
$C_j \leftarrow \frac{C_{free}}{numflag}$
$C_{free} \leftarrow 0$

Fig. 5. Our QoS-based shared cache partitioning algorithm. The algorithm tries to maximize the number of applications whose QoS will be satisfied. If it is able to satisfy every application's QoS, it tries to maximize the overall cache hit rate. If the available cache is insufficient to satisfy the QoS, it will always try to give applications whose QoS is not satisfied an equal share of the remaining cache instead of penalizing such applications completely.

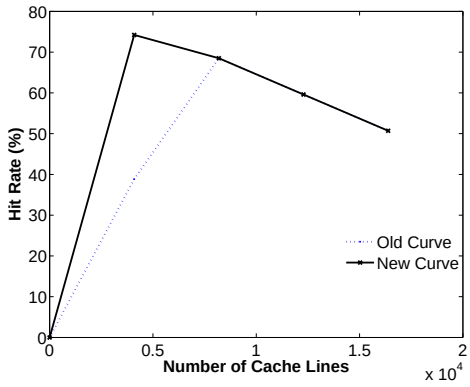


Fig. 6. Instance of curve fitting captured during the execution of tpc-c under a storage cache of 64 MB. The dotted line represents the interpolated hit rate-cache capacity curve. In the next iteration when a new data point is collected, the new data overwrites the old data so that when the interpolation is rerun, we obtain the solid line. The new data causes a shift in the curve. Thus, tpc-c now has recorded a higher hit rate for a smaller cache size, implying a decrease in the partition size required to satisfy its QoS.

linear relationship between two quantities. In fact, from Fig 2, the curve of hit rate versus cache size is exponential, namely, $hr \approx a(1-e^{-bC})$, where hr is the hit rate of the application, C is the corresponding cache capacity, and a, b are some arbitrary constants that depend on the application characteristics such as data reuse, locality, access pattern, and prefetch policy. The interpolation used in the curve fitting converts a sparse and interspersed dataset into a regular dataset that can be used to predict the minimum storage cache size required to satisfy the QoS.

For instance, in Fig 6, the dotted line represents the interpolated hit rate-cache capacity curve obtained from 5 points. In the next iteration when a new data point is collected, the new data overwrites the old data so that when the interpolation is rerun, we obtain the solid line. The new data causes a shift in the curve. Thus tpc-c now has recorded a higher hit rate for a smaller cache size, implying a decrease in the partition size required to satisfy its QoS. If the QoS value specified by the user is 68%, the curve fitting will predict that it takes less than 5,000 cache blocks to achieve this target hit rate instead of the earlier value of nearly 8,000 cache blocks required to achieve the same target hit rate (the points used in Fig 6 are initial profile points, and their accuracy is not very important, as will be explained below). Another behavior that can be observed from the figure is that the hit rate of the application increases with decreasing cache size. This behavior can be explained by observing Fig 3(c), where we see that tpc-c's hit rate drops for the same cache size depending on the phase in which it is executing.

Profiling Step: The first step of our approach is called the *profiling step*, and its main purpose is to obtain initial data points to start predictions. While we target three initial points, this can be any number larger than two.

The profiling step is off-line; that is, applications are statically profiled before they are initiated on the system. If the target storage cache can hold Y blocks, we can profile the application to be initiated on three representative cache capacities – 1, $Y/2$, and Y blocks – and obtain three corresponding performance points (hit rates). These points constitute our initial entries in the performance table maintained for that application. Profiling to obtain initial points is the only static component of our approach. The rest of our approach is dynamic and modulates the cache allocations of individual applications such that their QoS requests are satisfied.

Arguably, the initial points collected may not be very accurate. This is not a major problem, however, as these are only initial data points to start predictions. As an application executes more and goes through several rounds of cache allocations (of different sizes), we can expect more accurate future predictions. Additionally, whenever we obtain a new cache hit rate for a cache size that is already present in the table, we replace the old data in the table with the new point (i.e., different hit rates under the same allocated capacity at different points in execution) so that the algorithm captures "phase changes" in addition to the "hit rate versus cache size" behavior of an application.

Another concern that may arise is that the application behavior is derived from a completely different phase of an application. However, applications are sampled regularly at every quanta (fixed). Even though applications have multiple phases, these phases typically last over several quanta, and hence the partitioning strategy is good while the phase of the application lasts. Also, when a phase transition takes place, the curve fitting is always collecting new data and adapts its partitioning accordingly, as explained above, to account for the new phase.

Repartitioning Step: The next phase uses curve fitting, which initially works by using the three profiled points as input and later utilizes all consequent points recorded during

the iterative profiling. The QoS specifications are fed into this phase. The curve fitting constructs an approximate hit rate curve using interpolation from these points. The curve obtained is a rough estimate of the real hit rate curve the application exhibits, as shown in Fig 2. Using this interpolated curve and the QoS specification, the algorithm then calculates the *minimum storage cache capacity* to satisfy the specified QoS for that application. To maximize the number of applications whose QoS will be satisfied, we sort these applications based on their minimum required cache capacities. We then assign the physical cache to each application starting with the one that requires the least cache capacity until we are left with no free cache blocks or until the QoS requirements of all the applications are satisfied. We assume that all the applications have the same priority. *Each time the repartitioning is performed, more hit rate points as well as more recent points are accumulated, increasing the precision of the curve fitting as well as the quality of cache partitioning.* The amount of storage cache allocated to the application might vary during every iteration. However, the partition size allocated by the algorithm fluctuates around the cache size where QoS is likely to be satisfied, which is the region of interest. During this repartitioning step, there are two cases where an application's cache requirements may not be satisfied. The first case occurs when the cache demanded by the application (as specified by the QoS) cannot be satisfied by the physical storage cache in the system at all. That is, even if we allocate all cache capacity to the application, it is not possible to satisfy its QoS requirement. The other case occurs when the QoS requirement of the application can be satisfied in theory, but when other applications are considered, we may not be able to allocate enough space to that application to satisfy its QoS. Our algorithm does not distinguish between these two cases. An application whose capacity requirement (based on its QoS) cannot be satisfied in the current round of partitioning is *flagged*, and we do not assign it any storage cache space (in this step). Instead we revisit these flagged applications later.

Handling Nonflagged Applications: We execute this step only when there are no flagged applications and still some free cache is available. Since we want to maximize the overall storage cache hit rate, our implementation distributes the free cache among *all* the applications in proportion to the projected gains that each application will exhibit in terms of the hit rates. That is, we give more cache space to the application that can contribute greatest to the overall cache hit rate. If N is the number of concurrently executing applications, hr_j is the storage cache hit rate of the application j for the minimum QoS predicted using curve fitting, $max(hr_j)$ is the maximum recorded hit rate of application j, and C_{free} is the free cache space available for distribution, then each process gets $C_{free} \times \frac{max(hr_j) - hr_j}{\sum_{j=1}^{N}(max(hr_j) - hr_j)}$ blocks of the free cache. We base this premise on the fact that if an application has shown a higher hit rate in the past and even though its current hit rate may be lower, we are likely to see an increase in the hit rate of this application if we give it more cache. This can be observed from Fig 3(c), where we observe a peak in the hit rate of the application initially and then notice that the application hit rate goes down.

Handling Flagged Applications: During this step, if any free cache is still available, we distribute the free cache equally *among all the flagged applications only*. Since it is difficult to satisfy the QoS of these applications, we chose to give them an equal share of the cache rather than trying to maximize the hit rate of these applications.

We give two example scenarios to illustrate how our scheme operates. Let us assume that three applications use the storage cache and that currently these three applications occupy 40, 30, and 30 blocks respectively of the 100 blocks of cache space. Assume further that after the next round of curve fitting, we determined that the specified QoS for these applications can be satisfied by allocating 15, 30, and 30 blocks of the cache space. This means that while the second and third applications can maintain their cache allocations, we can take away 25 blocks of cache space from the first application and redistribute it. We measure the gain of an application by calculating the difference between its current hit rate and its maximum recorded hit rate. For instance, if the current hit rates of these three applications are 70%, 90%, and 90%, respectively, and the maximum hit rate recorded for these applications is 85%, 92%, and 95%, respectively, then we calculate the total of the difference between the maximum recorded hit rates and the current hit rates (i.e., (85-70) + (92-90) + (95-90) = 22) and then redistribute the remaining 25 blocks of free cache space in the ratio $25 \times \frac{85-70}{22} = 17$, $25 \times \frac{92-90}{22} = 2$ and $25 \times \frac{95-90}{22} = 6$ among the three applications, respectively.

Suppose now that the required minimum cache partitions to satisfy the specified QoS values are 80, 70, and 40 cache blocks for the three applications, respectively (assuming that the current partitions are 40, 30, and 30 cache blocks in that order). In this case, clearly the QoS of all the applications cannot be satisfied simultaneously because we are deficient by 90 cache blocks. Our scheme handles this case as follows. We sort these applications in ascending order of their minimum cache requirements. In other words, we will begin assigning cache to the third application, then the second, and then the first application. This approach is taken in order to maximize the number of applications whose QoS will be satisfied. However, we notice that after assigning 40 cache blocks to application three, we are unable to satisfy the QoS of applications one and two and are left with 60 free cache blocks. We distribute these remaining cache blocks to applications one and two equally; that is, each gets 30 cache blocks. The first, second, and third applications are now allocated 30, 30, and 40 cache blocks, respectively.

IV. Base Cases

We now explain the two base schemes against which we compare our *QoS-aware storage cache partitioning policy*.

TABLE I
MAJOR SIMULATION PARAMETERS.

Parameter	Value
Processors	1
Number of Disks	1
Disk Capacity	9.1 GB
Disk RPM	10,045
Disk Seek Time	5 msec
Partitioning Interval	2 sec simulation time
Page replacement policy	LRU
Cache block size	4KB

TABLE II
WORKLOADS (MIX TRACES) USED IN OUR EXPERIMENTS.

Workloads	Application
I	tpc-c, tpc-h
II	lu, tpc-h
III	mplayer, tpc-c
IV	lu, mplayer, tpc-h
V	lu, mplayer, tpc-c, tpc-h
VI	tpc-c, tpc-c
VII	lu, lu

A. No-Partitioning

In the no-partitioning scheme, the entire cache is shared among all competing applications as in Linux. There is no means of providing isolation or QoS guarantees in this scheme. Applications having poor data locality and high I/O rates tend to push out the working sets of applications with good locality and small working sets. This scheme has been used traditionally by most systems because of its ease of implementation. However, this scheme cannot provide any isolation or service guarantees to applications.

B. Equal-Partitioning

In the equal-partitioning scheme, the storage cache is divided equally among competing applications. Thus, this scheme is able to provide isolation. However, it cannot adjust to the dynamic behavior of applications. Moreover, this scheme has the disadvantage that applications requiring more cache might end up getting lesser cache, and their hit rates may suffer considerably. Likewise, an application that has little or no reuse and that requires very little cache space might end up getting much more cache space than required, without substantial boost in the hit rate. As a result, the hit rates of other applications go down and the overall cache hit rate suffers. This scheme may or may not always increase the overall cache hit rate, depending upon the combination of simultaneously executing applications and their cache space requirements.

V. EXPERIMENTAL EVALUATION

A. Setup

To test the effectiveness of our algorithm, we used the AccuSim simulator [21] and augmented it with our QoS-aware partitioning algorithm.[5] AccuSim is a trace-driven simulator used to simulate the kernel buffer cache [21], [13]. The traces used to drive the simulator are collected from live applications running on Linux using the strace utility. Traces captured contain information such as the process id, inode, size of I/O block, and type of I/O access (seek, read). AccuSim simulates I/O of an application accurately by interfacing with the Disksim simulator [22]. It simulates computation by recording the difference in time between successive I/O calls. AccuSim also simulates the default Linux prefetching policy. We turned on I/O prefetching in the simulator for all our experiments. For our experiments, we maintained 1,024 entries for every application's performance table (cache capacities and their corresponding hit rates). All major simulation parameters used are shown in Table I.

To simulate the different applications running simultaneously, we created *workloads* (mixes of traces) from the individual traces. Table II shows the workloads used in this study, and Table III gives a summary of the applications used. We used the above applications for the diversity that they exhibit in their access patterns, which range from sequential to random accesses and low reuse to high reuse. Later, we also present results for uniform workloads. As a consequence of increasing the hit rate of an application, the application might run faster, thus causing contention on the path from CPU to main memory. In our experiments, we assume that the bandwidth from storage cache to the higher level caches/CPU is infinite. Modeling the contention resulting from constrained bandwidth from storage cache to CPU is beyond the scope of this paper.[6]

B. Results

Recall that our goal is to maximize the overall storage cache hit rate while satisfying QoS of all the applications. In the first set of experiments, we use workloads I, II, and III. Fig 7 summarizes the overall and individual hit rates of the applications that are running concurrently. The QoS values used in our experiments are noted in the caption of the figure. We show the results obtained for each workload for the smallest cache size that is able to satisfy the QoS. As is evident from these plots, our proposed QoS-based cache partitioning scheme is able to adapt the partition sizes such that with a negligible drop in hit rate of one application (in Figs. 7(a) and 7(c) the drop in hit rate is 0.7% and 0.4%, respectively), it is substantially able to improve the hit rate of the other applications. In fact, not only is our scheme able to provide isolation among the two competing applications, but it is also able to achieve hit rates for most of the competing applications almost as high as the case where the application is the sole consumer of the cache.

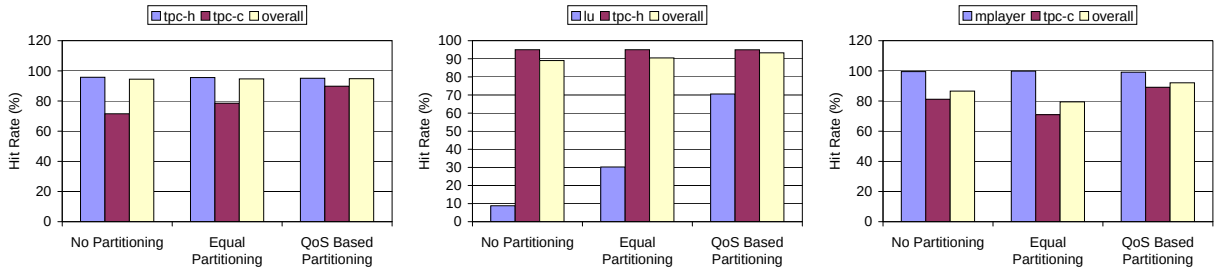
We note that our implementation has a very low software overhead. The history information maintained is fixed and is a tunable parameter whose size can be controlled. Additionally, we invoke our curve fitting every 2 sec of simulation time, which corresponds to several minutes of application runtime.

[5] Simulations have been used extensively in the past [1], [3], [4], [5], [6], [7], [8], [9] to verify several caching policies.

[6] When there are constraints however, the effects of contention to main memory will be negligible compared to the improvement brought about by reducing the disk accesses. Also, a good caching policy used for L1-L2 caches should help reduce the contention.

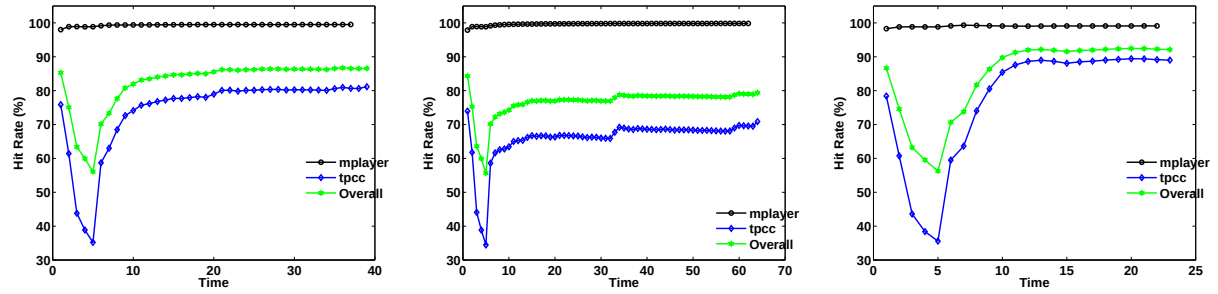
TABLE III
SUMMARY OF THE APPLICATIONS USED.

Application Description	Dataset Size	Disk Requests
LU: lu decomposition is a method used in linear algebra to factor a matrix as a product of a lower triangular matrix and an upper triangular matrix. LU decomposition is used in numerical analysis to solve a system of linear equations. We used the out-of-core (OoC) implementation from ScaLAPACK [18].	576 MB	1133571
Mplayer: mplayer is software used on Linux for playing movies. This benchmark was used to represent streaming applications. We used mplayer v1.0rc2 for our experiments. This category of applications typically exhibit sequential data accesses and hence have low temporal reuse and high spatial reuse.	1 GB	358922
TPC-C: The Transaction Processing Performance Council (TPC) [19] benchmark C is an on-line transaction processing (OLTP) application. It involves a mix of five different concurrent transactions of different types and complexity. TPC-C simulates a large environment in which users are executing transactions on a database. We used an open source implementation of tpc-c known as TPCC-UVa [20] that works with PostgreSQL v8.1.4 (another open source database).	137 MB	861320
TPC-H: tpc-h is a decision support benchmark from the OLTP suite of benchmarks. We used the tpc-h implementation provided by TPC with the open source database PostgreSQL v8.1.4. TPC-H exercises different ad hoc queries and concurrently modifies the database. The queries involve a huge volume of read and write requests. For our experiments, we exercised query 17.	1 GB	15150904



(a) Workload I (b) Workload II (c) Workload III

Fig. 7. Individual and overall hit rates of the applications in workloads I, II and III. (a) Workload I, 64 MB cache, $QoS_{tpc-h} = 94\%, QoS_{tpc-c} = 85\%$. (b) Workload II, 256 MB cache, $QoS_{lu} = 70\%, QoS_{tpc-h} = 94\%$. (c) Workload III, 64 MB cache, $QoS_{mplayer} = 98\%, QoS_{tpc-c} = 85\%$. Note that in the absence of partitioning, it would require caches of size 128 MB, 512 MB and 256 MB to satisfy the same individual QoS values specified by the user.

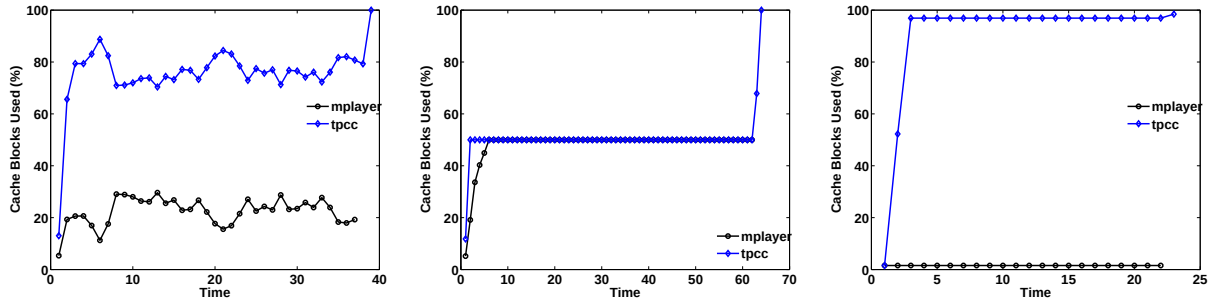


(a) No Partitioning (b) Equal Partitioning (c) QoS Based Partitioning

Fig. 8. Modulation of cache hit rate over time for workload III for a 64 MB cache, $QoS_{mplayer} = 98\%, QoS_{tpc-c} = 85\%$.

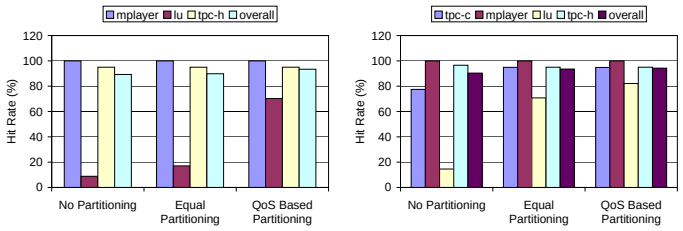
Also, since we use piecewise linear curve fitting, our scheme has a low computational overhead.

Fig 8 shows the hit rate variations over time for workload III for the no-partitioning, equal-partitioning, and our QoS-based cache partitioning cases. As can be seen from the figure, the overall and individual hit rates of the applications under the QoS-aware scheme not only satisfy the user-specified QoS but also are much better than the corresponding hit rates achieved by the no-partitioning and equal-partitioning schemes. Observe that the final cache hit rate of the applications achieves the target QoS values rather than the hit rate sampled every time quanta. We aim for a long-term satisfaction of QoS rather than trying to satisfy the QoS at every sampling instance. Thus, our scheme is more cumulative rather than reactive. Fig 9 shows the cache usage (number of cache blocks in use) of the individual applications for workload III over time for the different partitioning schemes. From the figure, we see that even though tpc-c dominates the cache usage, mplayer is causing interference in the cache, and tpc-c is unable to achieve its full potential hit rate in the presence of mplayer. Equal-partitioning is unable to provide any respite to tpc-c. Instead of assisting tpc-c in achieving a better hit rate, the cache usage of tpc-c goes down, and the overall hit rate of the storage cache suffers. We observe that our QoS-based adaptive scheme is able to detect and give more cache to applications that demand more cache to achieve higher hit rates rather than to applications that are able to sustain a high hit rate given a small cache allocation. We note from Fig 9(c) that our QoS-



(a) No Partitioning (b) Equal Partitioning (c) QoS Based Partitioning

Fig. 9. Storage cache usage over time for workload III for a 64 MB cache, $QoS_{mplayer} = 98\%, QoS_{tpc-c} = 85\%$.



(a) Workload IV (b) Workload V

Fig. 10. Individual and overall hit rates of applications for workload IV and workload V (a) Cache Size 256 MB, $QoS_{mplayer} = 98\%, QoS_{lu} = 70\%, QoS_{tpc-h} = 94\%$, (b) Cache Size 1024 MB, $QoS_{tpc-c} = 93\%, QoS_{mplayer} = 98\%, QoS_{lu} = 82\%, QoS_{tpc-h} = 94\%$.

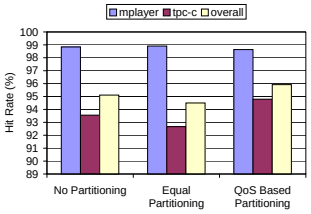


Fig. 13. Individual and overall hit rates of workload III with different QoS values for a cache size of 256 MB. $QoS_{mplayer} = 98\%, QoS_{tpc-c} = 94\%$. Refer to Fig 12(c) for results obtained using the old QoS values.

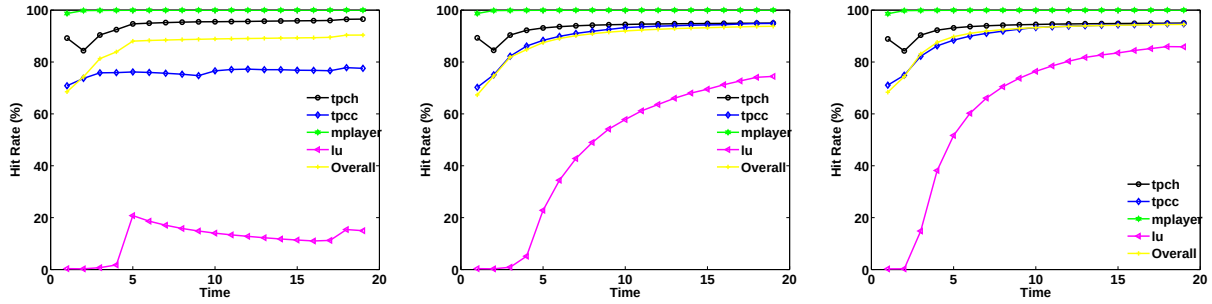
based scheme adapts quickly while deciding the application partitions. This is because our objective is to achieve the QoS values over a long period of time as opposed to some approaches that try to maintain QoS values per time quanta which we term as instantaneous QoS-based partitioning. Keep in mind that even though our QoS-based partitioning captures the twofold behavior of an application over time and over a range of cache sizes, we run the curve fitting on data collected for different cache sizes. Hence, our scheme always predicts the final size of the cache that will be required to achieve the QoS values, and therefore we see quick stabilization. *Observe that the x-axis of (a), (b), and (c) in Figs. 8 and 9 is different because it represents the overall execution time. Whenever there is an increase in the hit rate, the x-axis reflects the application speedup (decrease in runtime)*. This applies to all the results presented in this paper.

In the next set of experiments, we studied the effect on our algorithm of varying the number of simultaneously executing applications. We captured traces with three and four simultaneously executing applications. Figs. 10(a) and (b) show the individual hit rate and the overall hit rate of all applications in workloads IV and V, respectively. From the figure, one can see that our partitioning scheme supersedes the no-partitioning and equal-partitioning cases. From Figs. 10(a) and (b), we observe that our scheme maximizes the individual hit rates over the other partitioning schemes, resulting in improvement in the overall hit rates. Fig 11 shows the hit rate modulation over time for workload V. Our proposed scheme boosts the hit rate of tpc-c and lu by taking away cache space from the applications mplayer and tpc-h, which tend to pollute the cache with their large number of accesses.

We present a study of the effect of variation in parameters on our cache partitioning scheme. In these sensitivity experiments we explored (i) varying the storage cache size and keeping the QoS constant, (ii) varying the QoS while keeping the number of applications constant, and (iii) varying the QoS among multiple instances of the same application.

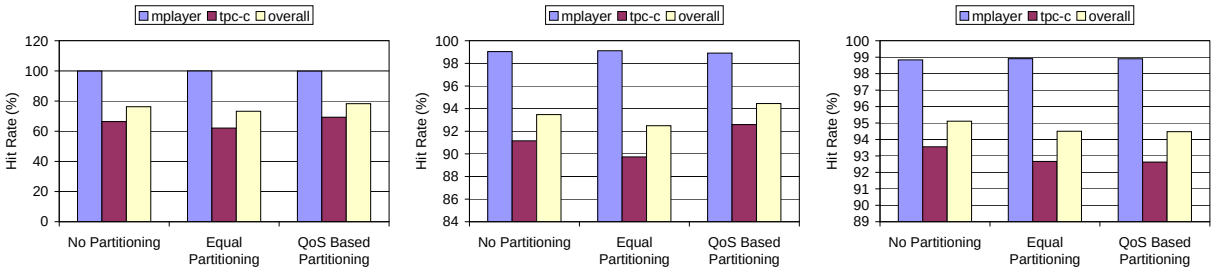
To understand the effect of varying the storage cache size while keeping the QoS constant, observe Fig 12. In this figure, we see a cache size that is too small to accommodate the QoS, as well as a cache that is too large. From the figure, we see that even if the cache is too small to accommodate the QoS values, our scheme still achieves the best hit rates possible. However, Fig 12(c) shows that if the QoS value is much below the maximum hit rate that is achievable by an application for the physical cache size, our algorithm will try to maintain the hit rate close to the QoS region and distribute the remaining free cache space proportionally among the applications, resulting in a hit rate comparable to the equal-partitioning case. Thus, we conclude that if the QoS is badly specified, the performance of our scheme is comparable to the equal-partitioning scheme.

In the next set of experiments, we kept the number of simultaneously running applications constant and varied the QoS. The old QoS values are specified in the caption of Fig 12 while Fig 12(c) shows the results obtained using the old QoS values. Fig 13 shows the results obtained for a 256



(a) No Partitioning (b) Equal Partitioning (c) QoS Based Partitioning

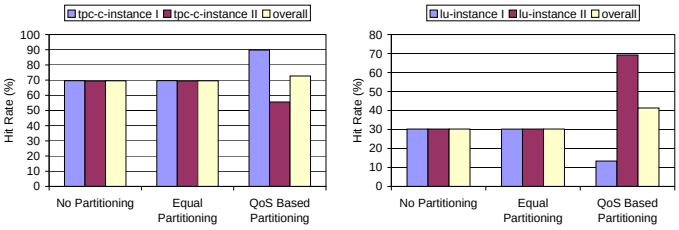
Fig. 11. Modulation of cache hit rate over time for workload V for a 1024 MB cache, $QoS_{tpc-h} = 94\%, QoS_{tpc-c} = 93\%, QoS_{mplayer} = 98\%, QoS_{lu} = 82\%$.



(a) 32MB cache (b) 128MB cache (c) 256MB cache

Fig. 12. Individual and overall hit rates of the applications for cache sizes of 32 MB, 128 MB and 256 MB for workload III, $QoS_{mplayer} = 98\%, QoS_{tpc-c} = 85\%$. Refer to Fig 7(c) to see the hit rate of workload III for a cache size of 64 MB for the same QoS values.

MB cache using the new QoS values. The results indicate that specifying a higher QoS value actually improves the individual hit rates. These results are consistent with the results obtained earlier where we vary the cache size instead of the QoS values. Next, we executed multiple instances of the same application and varied the QoS among the different instances of the application. We chose tpc-c and lu for these experiments. Fig 14(a) sums up the overall hit rate and individual hit rates of tpc-c when run concurrently and using different QoSs while Fig 14(b) shows the results of our partitioning scheme when using multiple instances of lu having different QoSs. As can be seen from the figure, the no-partitioning and equal-partitioning cases behave alike, and the two applications get similar hit rates. However, our scheme is able to distribute the cache among the two different instances so that each instance is able to satisfy its user-specified QoS. Additionally, we increase the overall hit rate of the storage cache while doing so.



(a) 32MB cache (b) 256MB cache

Fig. 14. Multiple instances of the same application running on the system having different QoSs. (a) 64 MB cache $QoS_{tpc-c-I} = 85\%, QoS_{tpc-c-II} = 50\%$, (b) 256 MB cache $QoS_{lu-I} = 5\%, QoS_{lu-II} = 68\%$.

VI. Related Work

Many researchers have explored shared storage cache partitioning designs that attempt to avoid conflicts among multiple applications [23], [1], [2], [24], [5], [8], [25]. Cao et al.'s LRU-SP algorithm [23], [1] partitions the buffer cache among multiple processes by using application disclosed hints. Karma [5] partitions a multilevel cache accessed by a single client using application hints in order to maintain exclusive caching. MC^2 [8] extends Karma and partitions a multilevel cache globally and locally by using application-supplied hints about the access pattern and frequency of access. Redline [25] partitions the memory pages among interactive tasks such that it assigns each task a budget consisting of 256 pages. Patterson et al. [2] divide the cache into three partitions to accommodate prefetched data, hinted reuse data and unhinted reuse data based on hints specified by an application to maximize the hit rates observed in the cache. However, they do not provide QoS guarantees to concurrently running applications in the system. In our work, we propose a cache partitioning scheme that determines the best partition sizes for multiple applications based on QoS using curve fitting. Our approach does not rely on application hints while guaranteeing QoS.

Thiébaut et al. [26] perform offline cache partitioning by finding a cache size such that hit rate derivatives are equal for all applications. Gniady et al. [13] and Kim et al. [14] partition the shared file system cache between multiple processes by detecting the access patterns. Jiang and Zhang [9] partition server buffers dynamically among the clients in accordance to their working set sizes. Gill and Modha [27] partition the cache dynamically among sequential and random I/O streams in order to reduce the read misses in the cache. Wachs et al. [17] partition the cache equally among competing applications with the aim of providing isolation and efficiency. Our approach differs from their work since we primarily consider QoS specified by the user and our scheme does not require prior knowledge of the access pattern of an application.

The approaches that come closest to our work are [10], [28], [29], [30]. Goyal et al. [10] partition the storage cache to provide QoS by maintaining a ghost buffer cache that contains a list of all recently evicted blocks. Ko et al. [28] use several feedback controllers based on proportional, integral, and derivative components as well as maintaining a shadow list of all disk blocks that have been accessed in the past. Lu et al. [29], [30] use digital feedback control theory in order to provide QoS to web users using a proxy web cache. Kelly et al. [31] modify the cache replacement policy LFU to account for weights in order to provide QoS to web cache users. Our approach to partition the cache and provide QoS guarantees to users is simpler, has less overhead, does not rely on maintaining a ghost buffer cache, and does not depend on any particular cache replacement policy.

VII. Conclusion

We have experimentally demonstrated the interapplication interference that takes place in the storage cache when multiple applications use it simultaneously. We explain the different kinds of solutions that have been proposed to this problem, which range from static to dynamic partitioning schemes. We further identify the inadequateness of a static partitioning scheme and explain the difficulty associated with implementing a good dynamic partitioning scheme. We propose a novel dynamic cache partitioning scheme that uses curve fitting. Our scheme is able to capture the dynamic modulations in application behavior over time as well as over different cache sizes. We showed the effectiveness of our storage cache partitioning scheme in reducing application execution latency by increasing the individual hit rates of the applications by 67% and 53% over the no-partitioning and equal-partitioning cases, respectively. Additionally, we were able to increase the overall storage cache hit rates by 11% and 12.9% over the no-partitioning and the equal-partitioning schemes, respectively.

Acknowledgment

This work is supported in part by NSF grants #0833126, #821527, #0720749, #0724599, #0621402, and #0444158 and by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] Cao, Felten, Karlin, and Li, "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling," *ACM Trans. Comp. Sys.*, vol. 14, no. 4, pp. 311–343, 1996.
[2] Patterson, Gibson, Ginting, Stodolsky, and Zelenka, "Informed prefetching and caching," in *SOSP*, 1995, pp. 79–95.
[3] Megiddo and Modha, "ARC: A self-tuning, low overhead replacement cache," in *FAST*, 2003, pp. 115–130.
[4] Zhou, Philbin, and Li, "The multi-queue replacement algorithm for second level buffer caches," in *USENIX*, 2001, pp. 91–104.
[5] Yadgar, Factor, and Schuster, "Karma: Know-it-all replacement for a multilevel cache," in *FAST*, 2006, pp. 169–184.
[6] Jiang, Ding, Chen, Tan, and Zhang, "DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities," in *FAST*, 2005, pp. 101–114.
[7] Wong and Wilkes, "My Cache or Yours? Making Storage More Exclusive," in *USENIX*, 2002, pp. 161–175.
[8] Yadgar, Factor, Li, and Schuster, "MC^2: Multiple clients on a multilevel cache," in *ICDCS*, 2008, pp. 722–730.
[9] Jiang and Zhang, "ULC: a file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches," *ICDCS*, pp. 168–177, 2004.
[10] Goyal, Jadav, Modha, and Tewari, "Cachecow: Qos for storage system caches," in *IWQoS*, 2003.
[11] Watson, *Contouring: A Guide to the Analysis and Display of Spatial Data*. Pergamon Publishers, 1992.
[12] Choi, Noh, Min, and Cho, "An implementation study of a detection-based adaptive block replacement scheme," in *ATEC*, 1999, pp. 18–18.
[13] Gniady, Butt, and Hu, "Program-counter-based pattern classification in buffer caching," in *OSDI*, 2004, pp. 27–27.
[14] Kim, Choi, Kim, Noh, Min, Cho, and Kim, "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references," in *OSDI*, 2000, pp. 9–9.
[15] Takagi and Hiraki, "Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches," in *ICS*, 2004, pp. 20–30.
[16] Zhou, Chen, and Li, "Second-level buffer cache management," *IEEE PDS*, vol. 15, no. 6, pp. 505–519, 2004.
[17] Wachs, Abd-El-Malek, Thereska, and Ganger, "Argon: performance insulation for shared storage servers," in *FAST*, 2007, pp. 5–5.
[18] D'Azevedo, "The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines," *Concurrency: Practice and Experience*, vol. 12, no. 15, pp. 1481–1493, 2000.
[19] "Trans. proc. perf. council," 2005, http://www.tpc.org/.
[20] Llanos, "Tpcc-uva: an open-source tpc-c implementation for global performance measurement of computer systems," *SIGMOD Rec.*, vol. 35, no. 4, pp. 6–15, 2006.
[21] Butt, Gniady, and Hu, "The performance impact of kernel prefetching on buffer cache replacement algorithms," *SIGMETRICS Perf. Eval. Rev.*, vol. 33, no. 1, pp. 157–168, 2005.
[22] Ganger, Worthington, and Patt, "The disksim simulation environment," *EECS Tech. Report*, vol. CSE-TR-358-98, 1998.
[23] Cao, Felten, and Li, "Application-controlled file caching policies," in *USTC*, 1994, pp. 11–11.
[24] Tomkins, Patterson, and Gibson, "Informed multi-process prefetching and caching," in *SIGMETRICS*, 1997, pp. 100–114.
[25] Yang, Liu, Berger, Kaplan, and Eliot, "Redline: First class support for interactivity in commodity operating systems," in *OSDI*, 2008.
[26] Thiébaut, Stone, and Wolf, "Improving disk cache hit-ratios through cache partitioning," *IEEE Trans. Comp.*, vol. 41, no. 6, pp. 665–676, 1992.
[27] Gill and Modha, "SARC: sequential prefetching in adaptive replacement cache," in *ATEC*, 2005, pp. 33–33.
[28] Ko, Lee, Amiri, and Calo, "Scalable service differentiation in a shared storage cache," in *ICDCS*, 2003, p. 184.
[29] Lu, Saxena, and Abdelzaher, "Differentiated caching services; a control-theoretical approach," 2001, pp. 615–622.
[30] Lu, Abdelzaher, Lu, and Tao, "An adaptive control framework for QoS guarantees and its application to differentiated caching," 2002, pp. 23–32.
[31] Kelly, Terence, Chan, Man, Sugih, Jeffrey, and Mackie-mason, "Biased replacement policies for web caches: Differential quality-of-service and aggregate user value," *SSRN eLibrary*, 1999.