A Scheduling Framework that Makes any Disk Schedulers Non-work-conserving solely based on Request Characteristics

Yuehai Xu
ECE Department
Wayne State University
Detroit, MI 48202, USA
yhxu@wayne.edu

Song Jiang
ECE Department
Wayne State University
Detroit, MI 48202, USA
sjiang@eng.wayne.edu

Abstract

Exploiting spatial locality is critical for a disk scheduler to achieve high throughput. Because of the high cost of disk head seeks and the non-preemptible nature of request service, state-of-the-art disk schedulers consider the locality of both pending and future requests. Though schedulers adopting the approach, such as the anticipatory scheduler, show substantial performance advantages, they need to know from which processes requests are issued to evaluate locality. This approach is not effective when the knowledge about processes is not available (e.g., in virtual machine environment, network or parallel file systems, and SAN) or the locality exhibited on a disk region is not solely determined by individual processes (e.g., in the case of cooperative process groups and disk array where requested data are striped).

We propose a light-weight disk scheduling framework that does not require any process knowledge for analyzing request locality. Solely based on requests' own characteristics the framework can make any work-conserving scheduler non-work-conserving, i.e., able to take future requests as dispatching candidates, to fully exploit locality. Additionally, we show how to effectively extend the framework to the disk array environment. Our design, *Stream Scheduling*, is prototyped in the Linux kernel 2.6.31. With extensive experiments of representative benchmarks, and in various environments such as the Xen virtual machine and the PVFS parallel file system, we show that the proposed scheduling framework can improve their performance by up to 3.2 times.

1 Introduction

While the hard disk has maintained exponential growth in capacity as a function of time, and sustained improvement in peak throughput, its random access performance, which is mainly determined by disk seek time, is increasingly a bottleneck. This makes the disk scheduler, which aims to minimize disk seeks by exploiting spatial locality in the requests, increasingly important to disk performance.

1.1 Non-work-conserving Disk Scheduling

Traditionally a disk scheduler such as CSCAN and SPTF chooses a request from those that have arrived and are pending in its dispatch queue and dispatches it to the disk. In a *work-conserving* mode, the scheduler must choose one of the pending requests, if any, to dispatch, even if the pending requests are far away from the current disk head position. The rationale for *non-work-conserving* schedulers, such as the anticipatory scheduler (AS) [16] and Completely Fair Queuing (CFQ) [1], is that a request that is soon to arrive might be much closer to the disk head than the currently pending requests, in which case it may be worthwhile to wait for the future request.[1] If such a request does arrive soon and the benefit of avoiding the long-distance disk seek outweighs the cost of idle waiting, the decision to keep the disk head in place may be justified. This is commonly observed when there are multiple processes concurrently issuing synchronous requests. For a request synchronously issued by a process, the scheduler can see its next request only after the request is served. Without a short waiting period the spatial locality of requests from such a process cannot be exploited. In this context the spatial locality refers to the fact that nearby disk locations are likely to be accessed by two consecutive requests within a short period of time. A process has strong locality if soon after its current request is completed, the scheduler will receive its next request for a location close to the current request. While the traditional scheduler selects a request for dispatching only from currently pending requests, a non-work-conserving scheduler, in essence, selects one from currently pending requests *and* future requests to exploit locality among synchronously issued requests.

[1]Descriptions of requests' statuses, such as "currently pending" or "future requests", are relative to the time when a scheduling decision is being made.

1.2 The Issues

To be effective, a non-work-conserving scheduler needs to predict how long it will take for the next nearby request to arrive—the strength of the process's locality—with reasonable accuracy, so that a decision can be whether to wait, and if so, for how long. To this end, existing non-work-conserving schedulers, such as AS and CFQ, group requests according to their issuing processes, analyze locality for each group, and make predictions for each process. While analyzing and utilizing locality in the context of process is an intuitive and convenient choice, there are three scenarios that challenge this practice.

First, if the requests to a limited disk region are from multiple processes, the locality, which is the basis for any scheduler to make scheduling decisions, is the result of these processes' combined I/O behaviors. This is especially the case when these processes coordinate to issue their requests. To determine whether the disk head should wait for a future request, the scheduler cares only about the probability for a nearby request to appear quickly, regardless of whether the request is from the same process. Limiting locality analysis to each individual process may underestimate the locality actually available to the scheduler and lose opportunity for seek reduction.

Second, in many important system settings process information is not available to the disk scheduler. For example, in the virtual machine environment only the scheduler in the host OS or VMM can actually dispatch I/O requests to the disk, on behalf of guest VMs where processes run and generate the requests. The scheduler in the host usually can only tell from which VM it receives a request but cannot distinguish from which process on a VM the request is issued. When there are multiple processes running on a VM, lack of such knowledge at the host would make non-work-conserving host scheduler less effective. In distributed or parallel file systems such as NFS and PVFS, the daemon at the file server receives requests from the clients and passes them to the disk scheduler without telling it which processes at the client side actually issued them. For another example, the SAN system and hardware RAID have internal disk schedulers that are critical to the systems' efficiency. The system interface for through which I/O requests are accepted usually does not include process information about request source.

Third, one of assumptions made by non-work-conserving schedulers is that it is solely the process that determines how long it will take for its next request to be issued. For this reason, *thinktime*, the time period between two consecutive I/O calls of a process, is treated as an attribute of the process and is estimated using the process's history information to predict when its next request will arrive. However, if the disk is a member of a disk array over which data are striped, the next several requests from the process might go to other disks in the array and may not be immediately scheduled for those disks. Consequently, the timing for this disk to see its next request from the process is determined not only by the process's thinktimes, but also by the data striping pattern on the array as well as the scheduling decisions made at the other disks. By mistaking the time period between two consecutive requests from a process for the process's thinktime, a disk's scheduler finds little opportunity for non-work-conserving scheduling. However, the fact is that by coordinating the scheduling of disks in the array, it is possible to reduce the time period so that waiting for the next request can still be beneficial.

1.3 The Challenges

To address these issues, we have to give up the assumption on the availability of process information. Specially, a scheduler is still expected to take future requests into account when making scheduling decisions, even without the process information, so that the most suitable request among both currently pending requests and future requests can be selected for dispatching. There are several critical challenges in achieving this objective.

First, if locality were to be explicitly analyzed for predicting timing and location of the next request, we have to group requests according to some criteria to track locality for each group of requests. However, without process information, for any artificial grouping method it would be hard to accurately predict whether a request would appear whose locality is stronger than any of currently pending requests. For example, a seemingly effective method is to divide the disk into different regions, either evenly or accordingly to request concentrations, and then track locality in each region. However, if the region were set too small, one process's synchronous requests could span multiple regions, which makes the arrival of the next request in a region too late and thus the locality in each region too weak. If the region were set too large, requests in a large disk area would be included for locality tracking, making the measured locality weak because of large inter-request distance. In both cases the scheduler may lose the opportunity to schedule future requests. In addition, region size may have to be dynamically adjusted according to changing request distribution on disk, making meaningful locality analysis yet more difficult.

Second, locality is relative. When there are pending requests relatively close to the current disk head, the scheduler must evaluate only the probability of requests of strong locality, and the relatively remote requests become less relevant. In contrast, if pending requests are relatively remote, even some not-very-close requests need to be included for locality analysis so as not to lose opportunity for higher disk efficiency. Therefore one must determine which requests should be included in an analysis adapting to the locations of pending requests. This would significantly add to the complexity and cost of such algorithms.

Third, for data striped on a disk array, even if think-times can be sufficiently short for I/O-intensive applications, the time gaps between two continuous requests seen at each disk can be too large to be exploited by non-work-conserving schedulers at individual disks. In this case the challenge is whether it is possible to reduce the time gaps by coordinating individual disks' scheduling so that it becomes worthwhile for a disk to wait for a future request. If the answer is yes, the question is how to know when there is such a potential before taking action for the coordination. As such an action usually entails postponing service of other applications' requests, it could cause excessive overhead and adversely affect performance if it did not produce the expected saving in disk seek time.

1.4 Our Contributions

In this paper we propose a light-weight framework that uses only requests' characteristics, specifically requests' arrival times and requested data locations, to turn any work-conserving scheduler into a non-work-conserving one. These request characteristics are readily available in any storage system and are employed in almost all disk schedulers. In summary, we make the following contributions.

First, instead of using the conventional method of direct analysis of locality to make a prediction about future requests, we propose to track the judicious actions, either waiting for future requests or seeking to a pending request, that should have been taken for greater disk efficiency. A judicious action is the one that helps improve disk efficiency, and may or may not have actually been taken in the prior scheduling. After observing a consistent pattern of judicious actions, our scheduling framework guides the scheduler to follow the trend in making its next decision. In the meantime, the framework retains the mechanism provided by the corresponding work-conserving scheduler for avoiding long delay or even starvation in its request service. The framework is simple, efficient, effective, and minimally intrusive to the work-conserving scheduler.

Second, we propose an efficient scheme for non-work-conserving scheduling for the disk array. To this end, we create a virtual disk corresponding to a disk array and apply our proposed framework on it to evaluate the potential benefit of coordinating scheduling across the disks for a particular stream of requests. When the evaluation is positive, coordinated scheduling of all disks is conducted to make it possible for scheduling of future requests to be profitable.

Third, we have implemented and evaluated the scheduling framework for single disks and for disk arrays, collectively named *stream scheduling*, in the Linux 2.6.31 and Linux software RAID MD. Our experiments on the prototype system with a variety of benchmarks demonstrate its significant performance advantages.

Section 2 of this paper details the design of stream scheduling. Section 3 presents an extensive experimental evaluation. Section 4 describes related work, and Section 5 concludes.

2 The design of *Stream Scheduling*

While a non-work-conserving scheduler is designed to select one request of the lowest cost from currently pending requests and future requests, a key technique in the scheduling is the effective comparison of costs for serving these two types of requests. Because future requests are not available for immediate dispatching, the scheduler keeps the disk idle for some period of time waiting for them if it decides to schedule a future request. Accordingly the cost for dispatching a future request is the sum of the wait time and the request's service time, while the cost of dispatching a pending request is just its service time. To effectively implement a non-work-conserving scheduler, there are two critical questions to answer: (1) how likely it is to see a future request whose cost is lower than that of the pending requests; and, (2) which future requests can be the candidates for selection. The answer to the first question determines whether a future request should be selected—whether the disk should wait—and the answer to the second question determines the threshold of the wait period beyond which no requests would be qualified. In the proposed framework it is the stream scheduling algorithm that answers the two questions by taking three inputs, namely request arrival time, arriving request location, and pending request location.

When a scheduler is ready to dispatch a new request the stream scheduling algorithm makes the decision on whether or not to schedule a future request. If yes, it will leave the disk waiting for an incoming request of relatively strong locality. Otherwise, it will dispatch a pending request selected by the working-conserving scheduling algorithm. As the stream scheduling algorithm makes its decisions independently of the working-conserving scheduling algorithm, the scheduling framework is applicable to any working-conserving scheduling algorithms.

2.1 The Stream Scheduling Algorithm

We consider a decision to make the disk wait for future requests a judicious one if there exists a future request R such that $wait\_time(R) + service\_time(R) < service\_time(selected\_pending\_request)$, where $wait\_time(R)$ is the time period from the time when the decision is made to the time when request R arrives, $service\_time(R)$ is the time spent to serve request R, the first dispatched future request after the decision is made, and $service\_time(selected\_pending\_request)$ is the service time for request selected by the work-conserving scheduling algorithm when the decision is made. If the inequality does not hold, the decision that demands immediate dispatching of a pending request is a judicious one. Note that the evaluation of the inequality cannot be completed until a future request satisfying the inequality actually arrives or until $wait\_time(R) \geq service\_time(selected\_pending\_request)$ becomes true. To evaluate the inequality, the service time of a known request can be estimated according to the distance between the location of its requested data and current disk head position, which can be considered to be the location of the most recently served request [14, 16]. Therefore, no matter whether request $selected\_pending\_request$ is actually dispatched, $service\_time(selected\_pending\_request)$ can be estimated.

In the inequality only $service\_time(selected\_pending\_request)$ is known when the decision is being made, while $wait\_time(R)$ and $service\_time(R)$ are unknown. Generally there are two methods to predict whether the inequality will hold. One is the method adopted by existing non-work-conserving schedulers, which use wait times and service times of previous requests that belong to the same process to predict these two times for the next request from the process, respectively. This method does not work when the process information is unavailable, because we do not know which previous requests and which future requests should be included in the evaluation of the inequality. To address the issue we propose the second method, which identifies a series of recently served requests for which the inequality held to form a so-called *stream*. A stream of sufficient size indicates that it is likely that the inequality would continue to hold and a judicious decision is to wait for future requests.

Figure 1 illustrates how a stream is formed and how it is used for request scheduling. The figure shows the arrival and completion times of requests as well as the requests' positions on the disk in terms of their requested data's LBNs (Logical Block Numbers). When the scheduler is notified that a request is completed is the time for the scheduler to select one request from currently pending requests and eligible future requests, or requests satisfying the inequality. As we can see, the positions of pending requests determine the eligibility of future requests. This is what we expect. If there are nearby pending requests, the criteria to schedule a future request must be more strict to make it profitable. Otherwise, it may be affordable for the disk to wait for a longer time and/or for a request with longer distance to the recently completed request. We may not come to a conclusion on whether a future request should be selected, or whether the scheduling decision is judicious, until $service\_time(selected\_pending\_request)$ after the decision is made. Note that the conclusion does not depend on what the actual decision is. If later on we do find a request arriving at a time and a position that satisfy the inequality, this request is called the *child* of the recently completed request. Therefore, for a request that is highly likely to have a child, the scheduler should wait for the child request, instead of immediately dispatching a pending request. To predict whether a recently completed request would have a child, we introduce the concept of *stream*, which is a sequence of requests $[R_0, R_1, ..., R_{n-1}]$ that have arrived in time-ascending order. For any two adjacent requests (R_{k-1}, R_k) in the stream, R_k is the child of R_{k-1}. If the length of the stream is equal to or greater than a predefined threshold *stream_threshold*, the stream is considered *established*.

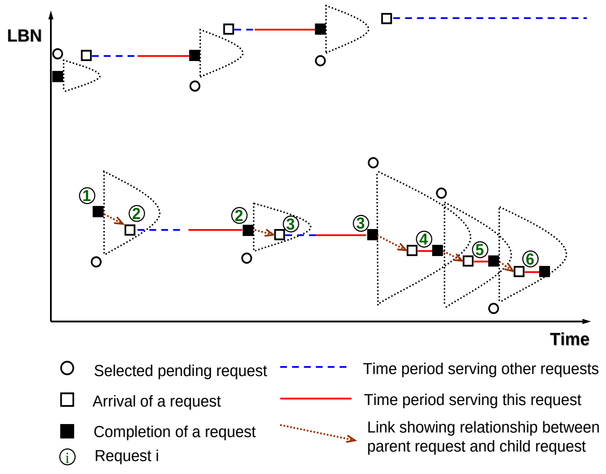
LBN
Time
1
2
2
3
3
4
5
6
Selected pending request
Arrival of a request
Completion of a request
Request i
Time period serving other requests
Time period serving this request
Link showing relationship between parent request and child request

Figure 1: Illustration of forming a stream and using the stream for scheduling. In the figure, the mushroom-shaped area ahead of each completed request describes the inequality on the eligibility of being a child request. The size of an area is determined by how close its corresponding pending requests are from the completed request. When a new request arrives in such an area, it becomes the child of the completed request associated with the area and extends the corresponding stream. As shown in the graph, the arrival of request 2 in the area following request 1 extends the stream to [1, 2]. When request 2 is completed, its area is created and the arrival of request 3 in the area further extends the stream to [1, 2, 3]. A stream cannot be established without new requests arriving in the defined areas, as shown in the upper part of the figure. In the lower part of the figure, before the stream is established, the disk head must leave and then seek back to serve its next request. When request 4 becomes a child request and joins the the stream, the stream is established (assuming that *stream_threshold* is 4). After this, the disk keeps serving requests in the stream (such as requests 5 and 6) for some period of time for high I/O efficiency.

The assumption we make in the stream scheduling algorithm is that for an established stream $[R_0, R_1, ..., R_{n-1}]$ ($n \geq stream\_threshold$), request R_{n-1} is highly likely to have its child request R_n extend the stream. The child request is the first one that arrives after the completion of R_{n-1} and satisfies the inequality, and the the disk should wait for the child request. This assumption is consistent with those made by other non-work-conserving algorithms to estimate thinktime and seek time of a process's next request. In addition, as we do not independently predict these two times, we can take the relationship between pending requests and future requests into account in the assumption. A disk waiting for a child request will stay idle for at most *service_time(selected_pending_request)* if there exist pending requests. The time when *service_time(selected_pending_request)* passes a request's completion time is called the request's deadline. After its deadline, it is not possible to find an eligible request to be the request's child. If the most recent request in a stream fails to find its child request, the stream aborts. Pseudo code for the algorithm is shown in Figure 2.

As shown in the pseudo code, when a request is completed it is possible for it to become a parent of a future request. So we insert the request into the *parent-to-be* queue to see if it would have a child that turns it into a parent. The queue is sorted by requests' deadlines, and only requests whose deadlines are not yet passed remain in the queue. Therefore, the size of the queue is usually very small. If the recently completed request is at the head of an established stream, we let the disk wait for a future request and in the meantime activate a timer for the completed request. Note that the algorithm does not remember every member of a stream. Instead, it only needs to keep track of the most recent request of a stream as well as its current length. When a new request arrives, we examine requests in the *parent-to-be* queue to see if it can extend a stream. If a request in the queue reaches its deadline without seeing a new request as its child, the stream led by the request is usually abandoned. One exception is that when stream has been sufficiently long—when its size is larger than *stream_threshold* by a factor of *tolerance_factor*, or 50% by default—we give the stream a second chance to get extended. When the disk has kept serving a stream for more than a threshold time period (*stream_time_slice*), the disk will dispatch a selected pending request, instead of waiting for a future child request in the stream (not shown in the pseudo code). In our work, we leave the issue of fairness to the external scheduler that has process information, or to the local work-conserving scheduler, such as the Deadline scheduler. When Deadline boosts the priority for dispatching of requests that have waited for too long the stream algorithm respects the decision by immediately sending them to the disk.

```
/* Procedure invoked upon completion of request R*/
R.completion_time = current_time;
R.position = LBN of data requested by R;

/* 'selected_pending_request' is the request selected
   by the work-conserving algorithm */
R.service_time =calculate_service_time(R.position,
                 selected_pending_request.position);
R.deadline = R.completion_time + R.service_time;

/* insert R into the queue sorted by requests'
   deadlines */
queue_of_parent_to_be <-- R;

/* If the stream is established, wait for a
   potential child request */
if (R.stream_size >= stream_threshold) {
   R.timer.timeout = R.service_time;
   activate R.timer;
} else
   dispatch selected_pending_request;

/* Procedure invoked upon arrival of request new_R*/
new_R.arrival_time = new_R's arrival time;
new_R.position = LBN of data requested by new_R;

for each request R in 'queue_of_parent_to_be' {
   if (R.deadline < current_time) {
     remove R out of the queue;
     continue;
   }
   if (new_R.arrival_time-R.completion_time+
   calculate_service_time(R.position, new_R.position)
   < R.service_time) {
     /* new_R is R's child */
     if (R.stream_size >= stream_threshold) {
       turn off R's timer;
       dispatch new_R;
     }
     new_R.stream_size = R.stream_size + 1;
     remove R from queue_of_parent_to_be;
     return;
   }
}
new_R.stream_size = 1;

/* Procedure invoked upon expiration of
   request R's timer */
if (R.stream_size >=
           (1+tolerance_factor)*stream_threshold){
  R.timer.timeout = R.service_time*tolerance_factor;
  R.service_time *= (1+tolerance_factor);
  R.deadline = R.completion_time + R.service_time;
  R.stream_size = stream_threshold;
  activate R.timer;
} else
  remove R out of 'queue_of_parent_to_be';
```

Figure 2: Stream scheduling Algorithm. In the pseudo code, function *calculate_service_time(disk_pos, req_pos)* is used to calculate the service time when the disk head is at *disk_pos* and the requested data is at *req_pos*, all in terms of LBNs. While we remember only the most recent member request of a stream and the size of a stream, we treat the size as an attribute of the request, denoted as *R.stream_size*. Enforcement of *stream_time_slice* is not included in the code.

The forming of streams and scheduling of requests are two independent procedures. That is, no matter what the scheduling decision is, the stream's development is not affected. The forming of streams is determined by the arrival and location of future requests, which usually do not depend on whether the disk actually waits for a child request, though the time period between a request's arrival and its completion is determined by the scheduling decision. Therefore, the stream scheduling algorithm can be used with any work-conserving scheduler. In addition, as the size of the *parent-to-be* queue is small, the algorithm is of low cost, specifically $O(N)$, where N is the size of the queue.

2.2 The Stream Scheduling Algorithm in a Disk Array

The effectiveness of non-work-conserving scheduling algorithms depends on the existence of locality in the requests of a process or a stream. This locality can be sufficiently strong to form an established stream when it is presented to the entire storage system. However, when the storage system consists of an array of disks where data are striped, each disk only sees a subset of the requests and the locality presented to individual disks can be much weaker. As each disk has to be individually scheduled to accommodate its specific data layout and request pattern, instead of all disks being fully synchronized and using one request scheduler [19, 8], it would be hard for each scheduler, on its own, to take advantage of the potential benefit of non-work-conserving scheduling. As an example, for a sequence of synchronous requests $[R_0, R_1, ..., R_{n-1}]$, which could be a stream if they were all served by a single disk, let us assume that only requests R_i (*i mod m* = *k*) reach disk k, where m is the number of disks in the array ($k = 0, 1, ..., m-1$). After serving R_0, disk 0 would not see R_m until $R_1, R_2, ...,$ and R_{m-1} have been served by other disks, whose service times depend on their respective scheduling decisions and could be significant if long-distance seeks are involved. Even worse, when one request has to access data spread on multiple disks, it is not completed until the last piece of the data is served, and the request's service time can be long if the disks are not coordinated to serve it quickly.

The time period between completion of a request and arrival of the next request of a stream observed at *one* particular disk (such as completion of R_0 and arrival of R_m at disk 0 in the example) consists of two types of time components. One is thinktime, or the time period from the completion of one request to the arrival of the next one of the stream observed by the disk array (such as completion of R_0 and arrival of R_1 in the example stream); another is response time, or the time period from the arrival to the completion of a request in the stream. A request's response time consists of its wait time and service time. To enable non-work-conserving scheduling, we need to minimize the time period for a disk to see its potential child request. While the involved thinktimes cannot be reduced for synchronous requests, the response time can be reduced by dedicating all disks to serving requests of a stream during a certain time period through disk coordination.

As we do not have process information, we set up a disk-array scheduler that treats the disk array as one big virtual disk and uses the method described in the stream scheduling algorithm to identify streams. The disk-array scheduler uses the array's logical addresses for calculating service times and uses pending requests on respective physical disks to evaluate the inequality for identifying child requests. The stream threshold for established streams is increased by m times, where m is the number of disks. Once a stream is established in the virtual disk, which we call a virtual stream, we attempt to find a stream on each physical disk corresponding to the virtual stream, which we call physical stream. Without dedicating all disks to the virtual stream, there is little chance for a physical disk to see its corresponding physical stream because of high response times. However, forcing all disks to serve only the virtual stream's requests before knowing whether the physical streams can be formed runs the risk of idling multiple disks for an excessively long time.

To address the challenge, we do not use a request's actual arrival time to determine whether it can extend a physical stream at a physical disk, as this time might be significantly reduced if all disks were dedicated to the corresponding virtual stream. Instead, we use the arrival time less the response times between the completed request and the disk's next request in the virtual stream (such as the arrival time of R_m minus the sum of response times of requests R_i ($1 \leq k \leq m-1$) in the example stream). The physical streams formed in this way represent the most optimistic estimates on future requests' arrival times, because the response times cannot be reduced to zero even if all disks are dedicated to the virtual stream. Once the array scheduler finds that physical streams have been established on all the disks for a particular virtual stream, it marks the virtual stream's next request to each disk as urgent so that it can be dispatched immediately to bring each disk head to the corresponding physical stream. After this, the array's scheduler instructs each disk's scheduler to use their respective physical stream for non-work-conserving scheduling and use the actual request arrival time to extend the stream. In this way, the non-work-conserving scheduling is certain to be cost-effective even though the physical streams are initiated with optimistic estimates of request arrival times. When a disk's physical stream is broken because it fails to find its next child request, this phenomenon usually cascades to other disks as it would cause other disks' streams to take longer time to see their respective next requests. When the array's scheduler observes broken physical streams, it will mark the virtual stream as *unusable*. Note the scheduler will keep maintaining the virtual stream to prevent a new stream from being formed and triggering non-work-conserving scheduling on the disks once again, which has been shown not to be cost effective. For the disk array, instead of letting each disk decide how long it continuously serves a physical stream, we let the array scheduler determine the time period during which each disk is supposed to serve its physical stream corresponding to the virtual stream. In this way the serving of requests in a virtual stream is fully coordinated across the disks.

3 Performance Evaluation

To evaluate the performance of the stream scheduling framework, we implemented it in the Linux kernel 2.6.31.3, either as a wrapper of a work-conserving disk scheduler to create a stream scheduler for individual disks, or as a revised implementation of the Linux software RAID *mdadm* for a disk array. In the experiments the CPU is an Intel Core2 Duo with 2GB DRAM memory and the disks are 7200RPM, 500GB Western Digital Caviar Blue SATA II (WD5000AAKS) with a 16MB built-in cache. The disk array has five disks connected to the host via a RAID card (RocketRAID 2320).

3.1 Disk Schedulers in Linux

Currently there are four configurable disk scheduler modules in the Linux distributions, each implementing a commonly used scheduler: Noop, Deadline, AS (or Anticipatory), and CFQ. Among them, Noop and Deadline are work-conserving while the other two are non-work-conserving. Noop simply dispatches a request as soon as it is received and does nothing beyond merging contiguous requests. Though it does not sound meaningful when the scheduler is used for dispatching requests directly to the hard disk, it is actually the preferred choice in other cases, such as in guest VMs of virtual machines and the systems using the SAN block device. This not only saves CPU cycles but also allows the requests to reach the lower level as early as possible, where a scheduler can see requests from different guest VMs or hosts and know how data are actually laid out on the disk(s) [32]. For this reason, we include Noop in the evaluation. Deadline is a scheduler approximating CSCAN augmented with a deadline-enforcement mechanism to prevent starvation. AS is a deadline scheduler enhanced with the anticipatory capability to wait for a future request that is of strong locality and is issued by the same process. CFQ aims to fairly distribute disk time among I/O-intensive processes and to bound request response time as Deadline does. As CFQ allows the disk to be idle waiting for future requests, it is non-work-conserving.

3.2 The Stream Scheduling in Linux

In the implementation we place Deadline in the stream scheduling framework and turn it into a non-work-conserving scheduler, the stream scheduler (SS). To accommodate the starvation avoidance mechanism, the stream scheduling algorithm respects the decision made by Deadline about immediate dispatching of expired requests by suspending its dedicated service to a stream. In the evaluation we set *stream_threshold* to be 4. We set *stream_time_slice* to 124ms if not stated otherwise, that is, a stream can be uninterruptedly served for at most 124ms if there are other pending requests in the system. This setting is consistent with that in AS for continuous requests from one process. We will present results of a sensitivity study on the parameter in Section 3.6.

Today's hard disks store multiple requests pending in it and enables its own scheduler such as NCQ for internal scheduling. The disk will continue serving requests pending in it after it completes a request. This poses a challenge to the implementation of the stream scheduling framework because the location of the most recently completed request is not necessarily the disk head position when the request it will dispatch next gets served. For example, when SS decides to idle the disk to wait for a future request by suspending dispatching requests, it assumes that the disk head will stay where it is. However, in a hard disk with stored pending requests, the disk head may have sought to another pending request scheduled by NCQ. To address the issue, we make a customization of the SS algorithm. In the kernel, there is a FIFO queue (*struct request_queue*), into which the disk scheduler dispatches its requests and from which the disk driver takes requests to the disk hardware. In other words, the actual service order will be basically consistent to the order in which the requests stay in the queue, assuming NCQ does not make a major change in the order. Accordingly, the disk head position when the next request is dispatched can be best indicated by the request at the queue tail, or the most recently inserted request. For this reason, SS makes a scheduling decision for the tail request when it is added into the queue, or considers it as the completed request in the stream scheduling algorithm, instead of for the actually completed request. If the decision is to wait for a future request, none of the currently pending requests are allowed to get into the queue and the corresponding timer will be activated at this time. In this way, the assumption made by SS about the disk head location still holds.

To estimate the service time of a request when the disk head is at *disk_pos* and the request is at *req_pos*, all in terms of LBNs (*calculate_service_time(disk_pos, req_pos)*), we adopted a simple empirical method which has been widely used for its effectiveness [25, 14, 16]. In this method, requests of various distances between two adjacent ones are sent to the disk and corresponding service times are collected. A smooth curve is fit through the measured [distance, time] data points and is used to represent *calculate_service_time()* function. In addition, as CSCAN prefers to serve requests in the forward direction, for the same inter-request distance we increase the cost of backward access by 50%.

3.3 Storage without Process Information

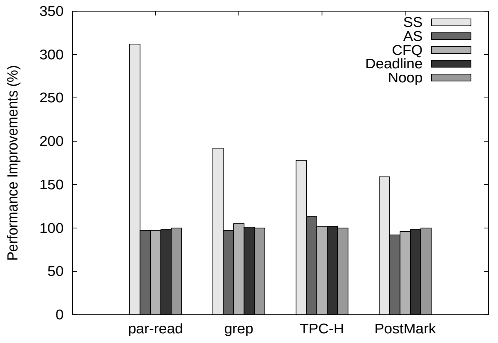

Figure 3: Performance of benchmarks *par-read*, *grep*, *PostMark*, and *TPC-H* with different disk schedulers (SS, AS, CFQ, Deadline, and Noop) when the process information of requests is removed from the workloads. The performance is presented as the schedulers' percentage improvement over that of Noop. For *par-read* and *PostMark* the performance is measured with throughputs, which are 16.0MB/s and 815.9KB/s, respectively for Noop. For *grep* and *TPC-H* the performance is measured with execution times, which are 73.5s and 228.2s, respectively, for Noop.

We first evaluate schedulers of storage systems for which process information for requests is not available, such as hardware RAID, SAN, and iSCSI connected storage devices. As the devices usually use proprietary software and their internal disk schedulers are not open-sourced for instrumentation, we hide process context information from the schedulers, or equivalently we make the schedulers believe that all requests are issued by the same process. In this section, we discuss the experimental results for one disk, and leave those for disk arrays to Section 3.5.

The benchmarks we use in this experiment are *par-read*, *grep*, *PostMark*, and *TPC-H*. *par-read* is a microbenchmark we wrote to study the impact of varying thinktime on the schedulers' performance. It creates four independent processes, each reading a 1GB file using 4KB requests in parallel. There is a 50GB gap between each two adjacent files. By default the thinktime between consecutive requests of a process is set to 0. *grep* is a Linux text search program we run to look for a non-existent word in the Linux 2.6.31 source code tree so that the entire directory tree is read. In the experiment we run two *grep*s, each reading one of two copies of the Linux directory with a 50GB gap between them. *PostMark* is to measure the performance of an Internet server running e-mail, netnews, or e-commerce applications, where random access of small files is the dominant access pattern [26]. In the experiment, we run four PostMark benchmarks (version 1.5.1), each creating a data set consisting of 10,000 files whose sizes are in the range between 0.5KB and 10KB. Each data set is 50GB away from the next data set. *TPC-H* is a decision support benchmark that processes business-oriented queries against a database system to examine large volumes of data. In our experiment we use PostgreSQL 8.3.7 as the database server and use DBT3 1.5.0 to create tables in it. We choose the scale factor 1 to generate the database and run query 19 against it. We run three *TPC-H* instances, with a 50GB space gap between adjacent data sets. Figure 3 shows the performance improvements of the four schedulers (SS, AS, CFQ, and Deadline) over Noop for the four benchmarks.

The experiments demonstrate that without process information both AS and CFQ lose the performance advantages they had enjoyed when they knew which requests are issued by the same process. Each process in the benchmarks synchronously issues its requests. For benchmarks *grep* and *PostMark*, which issue random requests and generally do not trigger prefetching in the operating system, the disk scheduler can see at most one request from a process at a time. Without seeing a nearby pending request, Deadline would dispatch a remote one and constantly move the disk head between remote data sets. This causes its performance to be as low as Noop. Without knowing which process actually issues a request, AS and CFQ assume all requests are from the same process and serve any pending requests when they see them, even if they are in distant regions. Consequently, they degenerate into work-conserving schedulers such as Deadline. However, if we let the information available to AS and CFQ in the experiments, they would perform as well as SS (with a performance difference less than 3%), demonstrating the importance of non-work-conserving scheduling.

Interestingly, the observations for random access can also be made on the other two benchmarks issuing sequential requests, which triggers prefetching in the operating system and allows the scheduler to see asynchronously issued requests. The condition for a work-conserving scheduler to keep serving one process's requests is to eliminate quiet periods in the process's I/O service, or the time period during which the scheduler does not see any requests from the process since last time when the scheduler attempts to

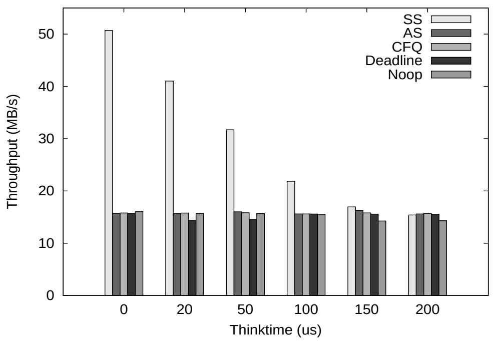

Figure 4: Throughputs of *par-read* with varying thinktimes, the time period between two continuous requests issued by a process.

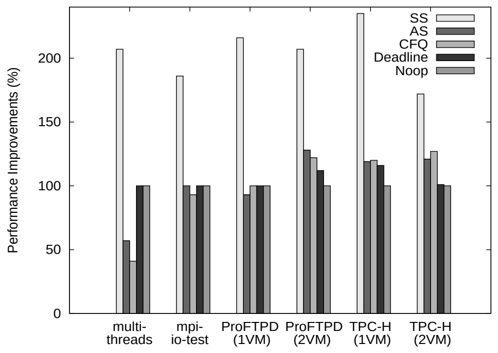

Figure 5: Performance of benchmarks *multi-threads*, *PVFS*, *ProFTPD*, and *TPC-H* with different disk schedulers. *ProFTPD* and *TPC-H* run either on one virtual machine or on two virtual machines. The performance is presented as the schedulers' percentage improvement over that of Noop. For *multi-threads*, *TPC-H(1VM)*, and *TPC-H(2VM)* the performance is measured with execution times, which are 65.7s, 231.4s, and 332.0s, respectively, with Noop. The performance of *PVFS*, *ProFTPD(1VM)*, and *ProFTPD(2VM)*, is measured with throughputs, which are 132.0MB/s, 17.1MB/s, and 12.5MB/s, respectively, with Noop.

dispatch this process's request. However, prefetching does not eliminate quiet periods in the system for two reasons. First, Linux maintains two readahead windows to prefetch file data. Prefetch requests issued for one window are contiguous and sent to the scheduler together. The scheduler has a good chance to merge them into one request. Consequently, the next prefetch request would not be triggered and sent to the scheduler until this request is completed and its data is consumed by the process. Second, as today's hard disks store multiple pending requests, a scheduling decision may have to be made before the process's request is completed. At this moment, it is likely the process's next prefetch request has not been generated, creating a quiet period. In both cases, Deadline, as well as AS and CFQ when process information is unavailable, would schedule other process's request and thrash the disk head among processes. While increasing the prefetch window can reduce number of quiet periods, they are unlikely to be fully removed. While SS does not rely on process information, its performance advantage is impressive with about 3.2X throughput improvement over the other schedulers. If we increase the thinktime, the performance improvement of SS becomes increasingly small as their wait times become larger (shown in Figure 4). When the thinktime is as large as $200\mu s$, the corresponding quiet periods increase to as large as about $8.5ms$, which causes streams to break and accordingly causes SS to stop waiting for future requests and behave like Deadline.

3.4 Storage with Inadequate Process Information

Next we consider four benchmarks running in an environment where the process information is inadequate or misleading. To investigate how synchronization of I/O-intensive threads affects behaviors of disk schedulers, we wrote a microbenchmark called *multi-threads*, in which there are four processes, each forking two threads. Each thread reads a 40MB file in a strided pattern, reading the first 4KB of data of every 16KB segment from the beginning to the end of the file. The distance between the two files accessed by one process is 100MB, and the distance of files read by adjacent processes is 50GB. Two threads of a process synchronizes after each makes every five requests. The performance improvements of the schedulers for the benchmark over that of Noop are presented in Figure 5. We can see that SS more than doubles the performance of Deadline in terms of reduction of execution time. Unfortunately AS and CFQ deliver performance even worse than that of Noop. The reason is that the synchronization disrupts their non-work-conserving scheduling, which is unnecessarily tied to the process. For example, assuming that two threads of a process are T_A and T_B, AS keeps serving requests from T_A by anticipatory wait until T_A reaches a synchronization point. Then AS has to wait for about $4ms$ until its timer expires and then it starts to serves T_B's requests, even though a T_B's request is pending nearby. In Linux a thread is presented as a light-weight process. Because the nearby pending request belongs to another thread, AS does not immediately dispatch it. Instead it suffers a long and unfruitful wait. In comparison, without relying on the process information SS is not constrained by the synchronization and dispatches any nearby requests.

PVFS is a parallel file system widely used in high-performance computing clusters [9]. We run the *mpi-io-test* program, an MPI-IO benchmark from the PVFS2 software package [30], on PVFS 2.8.2. The cluster has four compute nodes and eight data servers, where files are striped with a 64KB striping unit. Each data servers has a SATA disk (Seagate Barracuda 7200.10) with NCQ enabled. We run four such programs, each reading a distinct file with 10GB space in between. Each program has eight MPI processes, two per compute node, to read or write one 10GB file. The processes take turns reading 64KB blocks of data sequentially. For a particular data server, while requests from the same program have strong locality and SS can exploit the locality and achieve an improvement of aggregate throughput for all four MPI programs by 87% over Deadline or Noop, AS and CFQ seriously underperform (Figure 5). On each PVFS server there is a daemon called *pvfs2-server* accepting requests from compute nodes. To achieve asynchrony in its service, the daemon maintains a pool of threads and uses any available thread to dispatch its requests to the kernel. Consequently, AS or CFQ see requests associated with essentially randomly assigned thread numbers and can hardly recognize the locality within requests from the same thread, which leads to disk head thrashing among blocks of different files.

Xen is a virtual machine monitor that allows multiple guest virtual machines (VMs) to run on it [3]. In Xen, guest VMs send requests to their respective virtual block devices, which use the *blktap* mechanism to pass the requests to the kernel driver in the host VM, a privileged virtual machine that does the actual dispatch of I/O requests to disk. In the experiment we run two benchmarks, ProFTPD 1.3.1 and TPC-H, on Xen 4.0.1-rc6 to evaluate the disk scheduler in the host VM while leaving the schedulers in the guest VMs as Noop to quickly release requests into the host VM. ProFTPD is an FTP server [28]. In the test, we run a ProFTPD instance on each guest VM to serve four clients simultaneously downloading four 300MB files, respectively. There are 20GB space gaps between the files. For TPC-H, we use the same experimental setting for each guest VM as described in Section 3.3. From the experimental results shown in Figure 5 we see that SS significantly improves throughput, while AS and CFQ exhibit only limited, if any, improvements over Deadline and Noop because of their lack of process information about requests issued by processes on the same guest VM. When we run two guest VMs, each of the same setting as that in the one-VM scenario, AS and CFQ produce higher throughput improvement as they can differentiate requests from different guest VMs and thus reduce long-distance seeks among data requested by different VMs. Accordingly the relative performance advantage of SS is reduced.

3.5 Storage with Disk Array

To evaluate the performance impact of disk schedulers on the disk array, we select three benchmarks: *par-read*, *TPC-H*, and *PostMark*, whose settings are the same as described in Section 3.3, except that all files are striped over five disks with a 64KB striping unit. The disk array is organized as RAID0. We have also experimented with RAID5 and obtained consistent results. To focus on the performance challenges imposed by data striping on the disk array, we do not hide process information in the test. The experimental results are presented in Figure 6, which shows that for benchmarks of sequential access pattern, such as (*par-read* and *TPC-H*), SS achieves impressive improvements, 114% and 174% over that of Noop, respectively. Without opportunistic synchronization of the disks, the improvements made by AS or CFQ are limited. For example, AS reduces the execution time of *TPC-H* by only 25% while it can reduce the time by 72% when only one disk is used over that of Deadline (see the measurement in Figure 3 for SS, which produces about the same execution time as AS with known process information). The throughput of *par-read* with SS (361MB/s) approaches the peak throughput of the RAID card (around 400MB/s). The sequential access pattern with the help of aggressive prefetching in the RAID is turned into streams on each physical disk in SS, which helps eliminate disk thrashing. However, with the random access pattern of *PostMark*, SS shows minimal improvement as physical streams can hardly be formed.

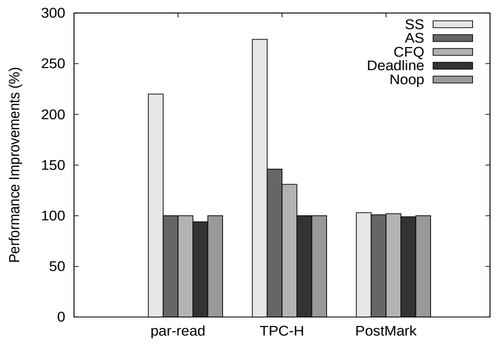

Figure 6: Performance of benchmarks *par-read*, *TPC-H*, and *PostMark*, with different disk schedulers in a 5-disk array. The performance is presented as the schedulers' percentage improvement over that of Noop. Performance of *TPC-H* is measured with execution time, which is 104.6s with Noop. For *par-read* and *PostMark*, it is measured with throughputs, which are 168.0MB/s and 1.3MB/s, respectively, with Noop

3.6 Impact of Stream Scheduling on Throughput and Response Time

SS achieves its performance advantage mostly through its dedication of disk service to one stream of requests during a certain period of time (*stream_time_slice*). By doing so, potentially long distance disk seeks take place only between time slices. Therefore, increasing the time slice is expected to reduce long-distance seeks and thus improve

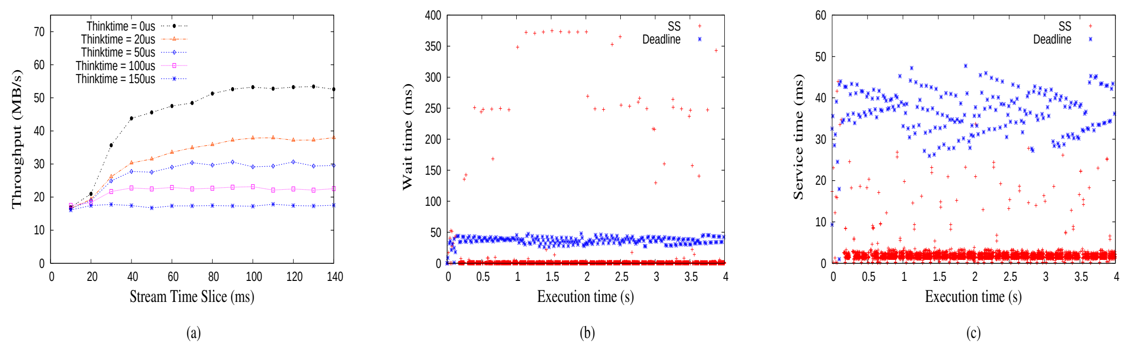

Figure 7: Impact of streaming scheduling on throughput improvement and variation of request response time. (a) Throughputs with varying stream time slices for benchmark *par-read* of different thinktimes. (b) Request wait times with SS of default time slice (124ms) for *par-read* of 0 thinktime with SS and Deadline. (c) Request service times with SS of default time slice (124ms) for *par-read* of 0 thinktime for SS and Deadline.

I/O throughput. However, requests that are pending but do not belong to the currently served stream may experience a longer pending period with increased time slice, which can increase variation in response time. To study the effect of the time slice on throughput and response time, we run benchmark *par-read* with an experimental setting the same as described in Section 3.3. As shown in Figure 7(a), the throughput improves with the increasing time slice. The more I/O intensive (with a smaller thinktime) the program is, the larger the improvement. The throughput improves quickly with I/O-intensive programs before the time slice reaches 100ms. After that, further increasing the slice yields only diminishing returns. This is why SS uses the default time slice of 124ms, the same value as adopted by Linux's AS. With this time slice, we measure two components of every request's response time, namely wait time and service time, during the execution of *par-read* with zero thinktime, and show them for the first four seconds of execution with SS and Deadline in Figure 7(b) and Figure 7(c), respectively. Unsurprisingly, SS produces some substantially large wait times (as large as 0.37s), as it rotates its service among four streams with a 124ms slice. Considering that Deadline's default timeout period for boosting request priority is 0.5s, these wait times are deemed acceptable. Meanwhile, as each cycle of such rotation produces only a few long wait times for synchronous requests, the percentage of requests with long wait times is very small and most requests have significantly reduced wait times with SS (Figure 7(b)). Furthermore, the use of a modest time slice in SS, which increases variation of response time, is paid off with significantly reduced request service time (Figure 7(c)) and improved disk efficiency.

4 Related Work

The effectiveness of disk scheduling is highly dependent on the existence of request locality. For this reason, there are many efforts to improve disk access locality. In the high-performance computing field many optimizations are made in the middleware to transform a large number of small non-contiguous requests into a smaller number of larger contiguous requests, including Data sieving [34], Datatype I/O [6], and Collective I/O [34, 43]. Because locality is about requested data locations on disk, there are many efforts to rearrange on-disk data layout to improve spatial locality, including data relocation [15] or data replication, either within one disk [14, 4, 20] or across multiple disks [42]. In addition, compiler techniques can be employed to improve locality by forming preferable I/O access patterns for the disks as well as optimizing file layouts matching known access patterns [18, 21]. However, the enhanced locality can be weakened or even lost when there are multiple processes, each concurrently issuing synchronous I/O requests. The locality can be recovered by non-work-conserving disk schedulers, such as the Anticipatory Scheduler [16]. Anticipatory scheduling has been implemented in some popular Linux disk schedulers including anticipatory [24] and (CFQ) [1].

The problem with the assumption by existing non-work-conserving schedulers on the availability of process information has been recognized in the literature, but effective solutions have not yet been proposed. One scenario is that the disk scheduler in the virtual machine monitor, such as AS, does not know from which specific process running on a guest virtual machine a request is issued. The Antfarm facility can help infer process information for disk scheduling by tracking activities of OS processes [17]. However, application of the technique is limited in the virtual machine environment. In addition, effort must be expended to implement the facility for each individual virtual machine system and the system must be open for instrumentation and patching. The difficulty caused by the lack of process information has also been found with the AS scheduler deployed in the NFS server [11], where the proposed approach is to use other access context information, such as accessed files' directory or owner, as hints to group requests for scheduling. While this approach can make up for the inadequacy to some extent, the hints may not be always relevant in revealing on-disk locality to the scheduler and could be misleading. A study of the Linux disk schedulers found that AS or CFQ can underperform significantly even when process information is available but multiple processes cooperatively send synchronous requests, because AS or CFQ may fail to find anticipation opportunity when it attempts to attribute history access statistics to individual processes [36]. By identifying access streams for non-work-conserving scheduling directly from the access locations, SS discards the requirement for process information instead of looking for its possibly inadequate substitutes with additional overhead in the OS or file systems.

The use of an I/O stream, or request sequence, to analyze and exploit access locality has been used before. Regarding I/O prefetching, though many sophisticated designs have been proposed, such as those based on probability graph model [38], information-theoretic Lempel-Ziv algorithm [7], or time series model [37], the stream-based approach dominates the design of prefetching in the system and has proven its effectiveness and efficiency [27, 41, 35]. Streams are also formed on the hard disk addresses to track disk access history and enable on-disk prefecthing [12]. Another interesting work is a tool called C-Miner that uses a data mining technique to find streams of disk block access representing repeatable block sequences, which can be used for initiating reliable prefetching [22]. While SS also tries to form streams among requests to the disk, the streams serve a different purpose. For prefetching, a well-established stream will lead to prefetching of multiple data blocks ahead of stream, while for SS the stream is maintained to determine whether the disk should wait for an upcoming request. More importantly, the cost of using streams in the aforementioned works can be much higher than that for SS when stream members have to be remembered for evaluation of stream quality, while SS needs only to track the latest member of a stream.

Regarding scheduling in the disk array, the necessity of coordinating requests has been widely recognized, especially for those with small striping units. When multiple disks are involved to serve a request, *"disks take different amounts of time to position, the request must wait for the slowest-positioning disk to transfer its data"* [10]. A possible solution is a synchronized interleaved disk system that synchronizes disk spindles and serves one request at a time in a disk array [19, 8]. However, for striping unit size larger than one byte or for a number of disks in a disk system beyond a certain limit, a fully synchronized disk array could seriously hurt performance by limiting the number of concurrently served requests [31]. The interference among requests from different processes caused by uncoordinated disk access has been reported and addressed in the cluster-based storage environment by using a timeslice-based co-scheduling method [40]. Though their work is similar to ours in the coordination of some or all disks and dedication of them to one process at a time, it cannot be effectively used as a disk scheduler to exploit spatial locality for higher performance. One reason is that their work requires an offline-calculated scheduling plan according to QoS specifications that does not adapt to the workload dynamics. Another reason is that it does not evaluate the benefits of dedicated service to a process relative to the cost of disk synchronization, and indiscriminately applies the synchronization to all programs. In contrast, SS dynamically evaluates the cost effectiveness of non-work-conserving scheduling by tracking and validating streams and opportunistically allows the disks to serve one virtual stream at a time. A scheme using opportunistic synchronization to reduce I/O interference among multiple MPI programs accessing a cluster of data servers has been proposed [44]. Without identifying streams, the scheme must assume a file is accessed by only one program and the MPI library and parallel file system must be instrumented to infer the assumed relationship and make it available to the scheduler. In contrast, SS provides a more general solution not constrained by availability of process information.

5 Conclusions

We have described the design and implementation of a stream scheduling framework that turns any work-conserving disk scheduler into a non-work-conserving one, even without process information available, to exploit locality embedded in the sequences of synchronous requests. The framework can also opportunistically coordinate the services at different disks of a disk array to recover and exploit the locality weakened by file striping. The framework has been prototyped in the Linux kernel, both as a disk scheduler and as a software RAID scheduler. Extensive experiments have demonstrated that SS can significantly improve the performance of representative benchmarks such as by TPC-H, PostMark, grep, FTP, as well as MPI programs. In particular, SS shows its unique value in environments where process information is unavailable, such as block or file storage servers and virtual machines.

6 Acknowledgements

We thank Kei Davis, Xuechen Zhang, our shepherd Elizabeth Varki, and the anonymous reviewers for their constructive comments that helped us to improve the paper. This research is supported by U.S. NSF CAREER award CCF-0845711 and NSF CRI-0708232.

References

[1] J. Axboe, "Completely Fair Queueing (CFQ) Scheduler," *http://en.wikipedia.org/wiki/CFQ*, 2010.

[2] D. Boutcher and A. Chandra, "Does Virtualization Make Disk Scheduling Pass?," *ACM SIGOPS Operating Systems Review*, Vol. 44, Issue 1, 2010.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *Proc. of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[4] 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 the 7th USENIX Conferenece on File and Storage Technologies*, 2009.

[5] A. Ching, A. Choudhary, K. Coloma, and W. Liao, "Non-contiguous I/O Accesses Through MPI-IO," *Proc. of IEEE International Symposium on Cluster, Cloud, and Grid Computing*, 2003.

[6] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp, "Efficient Structured Data Access in Parallel File System," *Proc. of IEEE International Conference on Cluster Computing*, 2003.

[7] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical Prefetching via Data Compression," *ACM SIGMOD Record Archive*, Vol. 22, Issue 2, 1993.

[8] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Surveys*, Vol. 26, No. 2, 1994.

[9] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel File System For Linux Clusters", *Proc. of the 4th Annual Linux Showcase and Conference*, 2000.

[10] P. M. Chen and D. A. Patterson, "Maximizing Performance in a Striped Disk Array,“ *Proc. of 17th annual international symposium on Computer Architecture*, 1990.

[11] H. Chen, J. Xiong, and N. Sun, "A Novel Hint-based I/O Mechanism for Centralized File Server of Cluster," *Proc. of IEEE International Conference on Cluster Computing*, 2008.

[12] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch," *Proc. of USENIX Annual Technical Conference*, 2007.

[13] G. Peng and T. Chiueh, "Availability and Fairness Support for Storage QoS Guarantee," *Proc. of IEEE International Conference on Distributed Computing Systems Conference*, 2008.

[14] H. Huang, W. Hung, and K. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption", *Proc. of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[15] W. Hsu, A. Smith, and H. Young, "The Automatic Improvement of Locality in Storage Systems," *ACM Transactions on Computer Systems*, Vol. 23, Issue 4, 2005.

[16] S. Iyer and P. Druschel, "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O," *Proc. of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[17] S. T. Jones, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "Antfarm: Tracking Processes in a Virtual Machine Environment," *Proc. of the USENIX Annual Technical Conference*, 2006.

[18] M. Kandemir and A. Choudhary, "Compiler-Directed I/O Optimization," *Proc. of the 16th International Symposium on Parallel and Distributed Processing*, 2002.

[19] M.Y. Kim, "Synchronized Disk Interleaving," *IEEE Transactions on Computers*, Vol. C-35, No. 11, 1986.

[20] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance," *Proc. of the 8th USENIX Conferenece on File and Storage Technologies*, 2010.

[21] M. Kandemir, S. Son, and M. Karakoy, "Improving I/O Performance of Applications through Compiler-Directed Code Restructuring," *Proc. of 6th USENIX Conference on File and Storage Technologies*, 2008.

[22] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou, "C-Miner: Mining Block Correlations in Storage Systems," *Proc. of 3rd USENIX Conference on File and Storage Technologies*, 2004.

[23] E. K. Lee and R. H. Katz, "An Analytic Performance Model of Disk Arrays and its Applications", *Tech. Rep. UCB/CSD 91/660, Univ. of California, Berkeley, Calif.*

[24] A. Morton, "Linux: Anticipatory I/O Scheduler", *http://kerneltrap.org/node/567*

[25] F. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Robust, Portable I/O Scheduling with the Disk Mimic," *Proc. of the 2003 USENIX Annual Technical Conference*, 2003.

[26] "The PostMark Benchmark", *www.freshports.org/benchmarks/postmark/*, 2010.

[27] R. Pai, B. Pulavarty, and M. Cao, "Linux 2.6 Performance Improvement through Readahead Optimization", *Proc. of the Linux Symposium*, 2004.

[28] "The ProFTPD Project", *http://www.proftpd.org/*, 2010.

[29] A. E. Papathanasiou and M. L. Scott, "Aggressive Prefetching: An Idea Whose Time Has Come," *Proc. of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[30] PVFS, http://www.pvfs.org/. Online-document, 2010.

[31] A. L. N. Reddy and P. Banerjee, "An Evaluation of Multiple-disk I/O Systems," *IEEE Transactions on Computers*, Vol. 38, No.12, 1989.

[32] Red Hat, Inc., "Oracle 10g Server on Red Hat Enterprise Linux 5 Deployment Recommendations," *http://www.redhat.com/*, 2008.

[33] E. Rosti, E. Smirni, G. Serazzi, Giuseppe, and L. Dowdy, "Analysis of Non-Work-Conserving Processor Partitioning Policies," *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1995.

[34] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[35] A. J. Smith, "Sequentiality and Prefetching in Database Systems," *ACM Transactions on Database Systems*, Vol. 3, No. 3, 1978.

[36] S. Seelam, R. Romero, P. Teller, and B. Buros, "Enhancements to Linux I/O Scheduling," *Proc. of the Linux Symposium*, 2005.

[37] N. Tran and D. A. Reed, "Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, Issue 4, 2004.

[38] V. Vellanki and A. Chervenak, "A Cost-Benefit Scheme for High Performance Predictive Prefetching," *Proc. of the ACM/IEEE conference on Supercomputing*, 1999.

[39] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance Insulation for Shared Storage Servers," *Proc. of the 6th USENIX Conference on File and Storage Technologies* , 2007.

[40] M. Wachs and G. Ganger,"Co-scheduling of disk head time in cluster-based storage," *Proc. of 28th International Symposium on Reliable Distributed Systems*, 2009.

[41] F Wu, H. Xi, and C. Xu, "On the Design of a New Linux Readahead Framework," *ACM SIGOPS Operating System Review*, Vol. 42, No. 5, 2008.

[42] X. Zhang and S. Jiang, "InterferenceRemoval: Removing Interference of Disk Access for MPI Programs through Data Replication," *Proc. of International Conference on Supercomputing*, 2010.

[43] X. Zhang, S. Jiang, and K. Davis, "Making Resonance a Common Case: A High-Performance Implementation of Collective I/O on Parallel File System," *Proc. of IEEE International Parallel and Distributed Processing Symposium*, 2009.

[44] X. Zhang, K. Davis, and S. Jiang,"IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-Server Coordination," *Proc. of Supercomputing*, 2010.