Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O

Sitaram Iyer Peter Druschel
Department of Computer Science, Rice University
{*ssiyer,druschel*}*@cs.rice.edu*

ABSTRACT

Disk schedulers in current operating systems are generally work-conserving, i.e., they schedule a request as soon as the previous request has finished. Such schedulers often require multiple outstanding requests from each process to meet system-level goals of performance and quality of service. Unfortunately, many common applications issue disk read requests in a synchronous manner, interspersing successive requests with short periods of computation. The scheduler chooses the next request too early; this induces deceptive idleness*, a condition where the scheduler incorrectly assumes that the last request issuing process has no further requests, and becomes forced to switch to a request from another process.*

We propose the anticipatory disk scheduling framework *to solve this problem in a simple, general and transparent way, based on the non-work-conserving scheduling discipline. Our FreeBSD implementation is observed to yield large benefits on a range of microbenchmarks and real workloads. The Apache webserver delivers between 29% and 71% more throughput on a disk-intensive workload. The Andrew filesystem benchmark runs faster by 8%, due to a speedup of 54% in its read-intensive phase. Variants of the TPC-B database benchmark exhibit improvements between 2% and 60%. Proportional-share schedulers are seen to achieve their contracts accurately and efficiently.*

1. INTRODUCTION

Disk scheduling has been an integral part of operating system functionality since the early days [7, 13, 15, 22, 34]. This paper examines disk scheduling from a system-wide perspective, identifies a phenomenon called *deceptive idleness* and proposes *anticipatory scheduling* as an effective solution.

Disk schedulers are typically *work-conserving*, since they select a request for service as soon as (or before) the previous request has completed [23]. Now consider processes issuing disk requests *synchronously*: each process issues a new request shortly after its previous request has finished, and thus maintains at most one outstanding request at any time. This forces the scheduler into making a decision too early, so it assumes that the process issuing the last request has momentarily no further disk requests, and selects a request from some other process. It thus suffers from a condition we call *deceptive idleness*, and becomes incapable of consecutively servicing more than one request from any process.

It is common for data requested by a process to be sequentially positioned on disk. Nevertheless, deceptive idleness forces a seek optimizing scheduler to multiplex between requests from different processes. The ensuing head seeks can cause performance degradation by up to a factor of four, as shown in the next section. In the related problem of proportional-share disk scheduling, meeting a given contract (i.e., a proportion assignment) may require the scheduler to consecutively service several requests from some process. Deceptive idleness precludes this requirement, thus limiting the scheduler's capacity to satisfy certain contracts. In both cases, the scheduler is reordering the *available* requests correctly, but system-wide goals are not met.

This paper proposes the *anticipatory disk scheduling framework*, and applies it to various disk scheduling policies. It solves deceptive idleness as follows: before choosing a request for service, it sometimes introduces a short, controlled delay period, during which the disk scheduler waits for additional requests to arrive from the process that issued the last serviced request. The disk is kept idle for short periods of time, but the benefits gained from being able to service multiple requests from the same process easily outweigh this loss in utilization. The framework is thus an application of the *non-work-conserving scheduling discipline*. The exact tradeoffs are sensitive to the original scheduling policy, so to determine whether and how long to wait each time, we propose adaptive heuristics based on a simple cost-benefit analysis.

We implement anticipatory scheduling as a kernel module in the FreeBSD operating system, evaluate it against a range of microbenchmarks and real workloads, and observe significant performance improvements and better adherence to quality of service objectives. For a trace-based, disk-intensive workload, the *Apache webserver* delivers between 29% and 71% more throughput by capitalizing on seek reduction within files. The synchronous, read-intensive phase of the *Andrew filesystem benchmark* runs faster by 54% due to seek reduction both between files and within each file;

consequently, the overall benchmark improves by 8%. Variants of the *TPC-B database benchmark* exhibit speedups between 2% and 60%: in the latter case, deviating from a standard TPC-B setup, by subjecting read-only queries to multiple separate databases leads to more seek reduction opportunities. The *Stride* proportional disk scheduler [32] achieves its assigned allocations even for synchronous I/O (assuming there is sufficient load), and simultaneously delivers high throughput.

After an exposition and analysis of deceptive idleness, we describe anticipatory scheduling in Section 3, and delve into a detailed experimental evaluation in Section 4. We discuss some emergent issues in Section 5, describe related work in Section 6, and conclude.

2. DECEPTIVE IDLENESS

This section describes and analyzes the phenomenon of deceptive idleness with two examples. In each case, the scheduler faces a shortage of the desired type of requests at critical moments.

Example #1: Seek reducing schedulers

Here is an example of how a seek reducing disk scheduler can degenerate to FCFS-like behaviour, and potentially suffer throughput loss by a factor of 4. Consider an operating system equipped with any seek reducing scheduler, like Shortest Positioning-Time First (SPTF) or CSCAN [34]. Let two disk-intensive processes p and q each issue several disk requests to separate sets of sequentially positioned 64 KB blocks. In the interest of seek reduction and throughput improvement, the scheduler would be expected to consecutively service many requests from p's set, then perform one expensive head repositioning operation, and service many of q's requests. This happens in practice, provided each process maintains one or more pending requests whenever the scheduler makes its decision: these moments in time are called *decision points.* Such an experiment results in a sustained throughput of 21 MB/s on our disk, owing to a service time of 3ms for every 64 KB block.

Now consider a scenario where the above requests are issued *synchronously* by the two processes, i.e., each process generates a new request a few hundred microseconds after its previous one finishes. A work-conserving disk scheduler never keeps the disk idle when there are any requests pending for service. It therefore tries to schedule some request immediately after (say p's) request has completed. At this decision point, process p has not yet been given the chance to perform the computation required to generate its next request. This forces the scheduler into choosing a request from q, performing a large head seek to that part of the disk, and servicing that request. The subsequent request for p arrives soon after, but disk scheduling is non-preemptible, and it is now too late to service this nearby request. This leads to the scheduler alternating in an FCFS manner between requests from the two processes. Throughput falls to 5 MB/s, due to 9ms of average seek time and 3ms of read time for every 64 KB block. The problem persists even if more than two processes issue synchronous requests. In this case, CSCAN degenerates to a round-robin scheduler, whereas SPTF alternates between some pair of processes.

Example #2: Proportional-share schedulers

This example shows how deceptive idleness can affect schedulers in ways other than degrading throughput. Consider a proportional-share scheduler like Yet-another Fair Queueing (YFQ) [7], Stride Scheduling [32], or Lottery Scheduling [31]. Their intended behaviour is to deliver disk service to multiple applications (e.g., processes p and q) in accordance with an arbitrary preassigned ratio. For an assignment of 1:2 (or 33%:66%), the scheduler may service a few requests for process p, and correspondingly, about twice as many requests for process q. However, if these processes maintain only one outstanding request at critical moments, then as in the previous example, the work-conserving scheduler is forced to alternate between requests from the two processes. It becomes incapable of adhering to the desired contract for this workload, and instead achieves proportions much closer to 1:1. Figure 1 shows results of such an experiment. This effect on proportional-share schedulers has been noted in [26] §5.6.

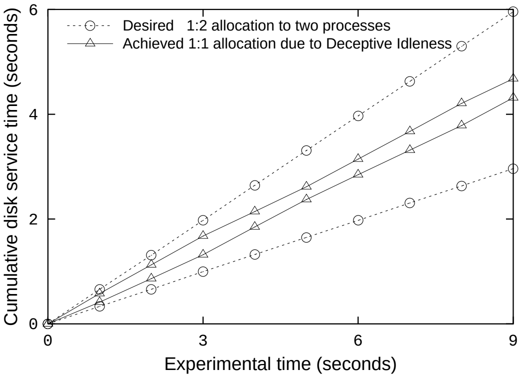


Figure 1: A proportional-share scheduler: The outer pair of lines denote ideal scheduler response to an allocation ratio of 1:2 to the two processes. Inner pair of lines: synchronous I/O causes requests to be almost alternately serviced from the two processes, yielding proportions much closer to 1:1.

If we have three active processes instead, say p, q, r with shares of 1:1:3, then Stride will be forced to schedule requests from processes in the sequence r, p, r, q, etc. and achieve the skewed proportions of 1:1:2. For nontrivial reasons, a lottery disk scheduler under similar circumstances will deliver proportions of 2:2:3 instead (see [14] §4.2.2 for details).

The underlying problem

In both examples, the scheduler reorders the available requests according to its scheduling policy, but fails to meet overall objectives of performance and quality of service. In essence, processes that issue synchronous requests cause the work-conserving disk scheduler to receive no requests from that process, in time for the following decision point.[1] This leads to deceptive idleness, rendering the scheduler incapable of exploiting spatial and temporal locality among synchronous requests.

[1] This happens despite the system-wide request queue generally being long and bursty on loaded server systems [22].

2.1 Prefetching

It is possible to partially work around the problem of deceptive idleness by using *asynchronous prefetch*. This involves predicting the future request issue pattern for a process, and issuing its immediately forthcoming request before the current one completes. Each process thus maintains multiple outstanding requests at decision points, and gives the scheduler the chance to service consecutive requests from the same process. Seek reduction opportunities can therefore be exploited if requests issued by a process are sequential. Likewise, a proportional-share scheduler would now have the capacity to adhere to its contract, even for synchronous requests.

Prefetch can be effected either explicitly by the application or transparently by the kernel. However, both approaches have fundamental limitations in terms of feasibility, accuracy, and overhead.

Application-driven prefetch

Applications can embrace programming paradigms and techniques that prevent the onset of deceptive idleness. They can use asynchronous I/O using APIs such `aio_read()` to prefetch future requests. Alternatively, they can roll their own asynchronous I/O using multiple processes or kernel threads, to proactively issue disk requests of the right type (e.g., sequential).

There are several problems with this. (1) Applications are often fundamentally unaware of their future access pattern, and may be incapable of issuing accurate prefetch requests. Examples include filesystem metadata and database index traversals, and predicting future requests in webservers. (2) Applications may have to be written in a cumbersome programming paradigm, whereas most applications are better suited to a sequential programming style. (3) Existing applications would have to be rewritten for this purpose, which may not be desirable or even possible under some circumstances. (4) Issuing explicit read requests using Unix API functions (instead of memory mapping a file) may entail more data copying and cache pollution, which could become expensive for in-memory workloads. Lastly, (5) the `aio_read` system call is an optional POSIX realtime extension, and may not be implemented or enabled in some operating systems.

Kernel-driven prefetch

Filesystems can (and most do) try to guess future request patterns for applications and issue separate asynchronous prefetch requests[2] for them. The usual reason is to overlap computation with I/O [24], but this prefetching also prevents deceptive idleness. There are, however, limitations to this transparent approach. The file system is typically even less capable of predicting future access patterns than applications are. Prefetch needs an exact notion of the location of the next request, and the penalties of misprediction can be high. This forces the prefetching to be complicated, yet conservative. Applications such as database systems can issue requests possessing spatial locality, but their access patterns may be extremely difficult to detect and effectively prefetch. Finally, sequentially accessed medium-sized files are often too small for the filesystem to detect sequential access and confidently issue prefetch requests. In summary, prefetching can potentially alleviate and even eliminate deceptive idleness, but limitations in its feasibility and effectiveness under many conditions discount it as a general solution.

Studies have shown an increasing trend in modern disk-intensive applications to issue non-sequential disk requests that nonetheless possess spatial locality [19, 30]. Prefetching has limited utility in these cases, and it is vital to consider complementary and more widely applicable alternatives.

3. ANTICIPATORY SCHEDULING

We now present a simple, practical, general, application-transparent and low-overhead solution to deceptive idleness. There are three necessary conditions for deceptive idleness to manifest itself: (a) multiple disk-intensive applications concurrently issuing synchronous disk requests, (b) the intrinsic non-preemptible nature of disk requests, and (c) a work-conserving disk scheduler, which schedules a request immediately upon completion of the previous request. Our solution takes the intuitive approach of eliminating condition (c), by wrapping a given disk scheduling policy in a non-work-conserving *anticipatory scheduling framework*.

When a request completes, the framework potentially waits briefly for additional requests to arrive, before dispatching a new request to the disk. Applications that quickly generate another request can do so before the scheduler takes its decision; deceptive idleness is thus avoided. The fact that the disk remains idle during this short period is not necessarily detrimental to performance. On the contrary, we will show how a careful application of this method consistently improves throughput and adheres more closely to desired service allocations.

The question of whether and how long to wait at a given decision point is key to the effectiveness and performance of our system. In practice, the framework waits for the shortest period of time over which it expects, in high probability, for the benefits of waiting to outweigh the costs of keeping the disk idle. An assessment of these costs and benefits is only possible relative to a particular scheduling policy: a seek reducing scheduler may wish to wait for contiguous or proximal requests, whereas a proportional-share scheduler may prefer weighted fairness as its primary criterion. To allow for such flexibility, while minimizing the burden on the developer of a particular disk scheduler, the anticipatory scheduling framework consists of three components: (1) The original disk scheduler, which implements the scheduling policy and is unaware of anticipatory scheduling; (2) a scheduler-independent *anticipation core*; and, (3) adaptive scheduler-specific *anticipation heuristics* for seek reducing and proportional-share schedulers.

Figure 2 depicts the architecture of the framework. The anticipation core implements the generic logic and timing mechanisms for waiting, and relies on the anticipation heuristic to decide if and how long to wait. This heuristic is implemented separately for each scheduler, and has access to the internal state of the scheduler. To apply anticipatory scheduling to a new scheduling policy, one merely has to implement an appropriate anticipation heuristic.

[2] different from synchronous readahead, where requests are enlarged to 64 KB to amortize seek costs over larger reads.

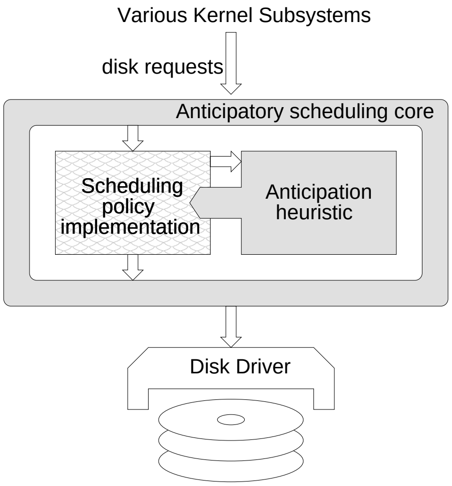

Various Kernel Subsystems
disk requests
Anticipatory scheduling core
Scheduling policy implementation
Anticipation heuristic
Disk Driver

Figure 2: Anticipatory scheduling framework

The remainder of this section spells out our two assumptions about workload characteristics, then describes the anticipatory scheduling framework, followed by appropriate anticipation heuristics for seek reducing and proportional-share schedulers. Finally it covers some implementation issues.

3.1 Workload assumptions

We tentatively make two assumptions about the granularity at which applications issue related disk requests.

Assumption #1: *Synchronous disk requests are issued by individual processes.* In most applications, a dependence between disk requests is explicitly reflected in code structure, so it is uncommon for multiple processes to coordinate to issue a set of synchronous requests. This assumption serves two purposes: (a) it considerably simplifies the anticipation heuristic, by requiring it to wait only for the process that issued the last request, and (b) it allows the anticipation core to optimize for the common case when this process gets blocked upon issuing a synchronous request.

Assumption #2: *Barring occasional deviations, all requests issued by an individual process have approximately similar degrees of spatial and temporal locality with respect to other requests from that process, and these properties do not change very rapidly with time.* The anticipation heuristic adaptively learns application characteristics on a per-process granularity; this assumption constitutes the basic requirement for adaptation to be possible. We therefore assume that a process does not interleave disk requests with markedly different locality properties.

Experimental results with real applications reported in Section 4 indirectly confirm that these assumptions hold to a sufficient degree. Relaxing these assumptions to accommodate a larger range of workloads is the subject of future work; some ideas in this direction are suggested in Section 5.2.2.

3.2 The anticipation core

A traditional work-conserving scheduler has two states, IDLE and BUSY, with transitions on scheduling and completion of a request. Applications can issue requests at any time; these are placed into the scheduler's pool of requests. If the disk is idle at this moment, or whenever another request completes, a request is *scheduled*: the scheduler's *select* function is called, whereupon a request is chosen from the pool and dispatched to the disk driver.

The anticipation core forms a wrapper around this traditional scheduler. Whenever the disk becomes idle, it invokes the scheduler to select a candidate request (as before). However, instead of dequeuing and dispatching immediately, it first passes this request to the anticipation heuristic for evaluation. A result of zero indicates that the heuristic has deemed it pointless to wait; the core therefore proceeds to dispatch the candidate request. However, a positive integer represents the waiting period in microseconds that the heuristic deems suitable. The core initiates a timeout for that period, and enters the new WAIT state. Though the disk is inactive, this state differs from IDLE by having pending requests and an active timeout.

If the timeout expires before the arrival of any new request, then the previously chosen request is dispatched without further ado. However, new requests may arrive during the waiting period; these requests are added to the pool. The anticipation core then immediately asks the scheduler to select a new candidate request from the pool, and asks the heuristic to evaluate this candidate. This may lead to immediate dispatch of the new candidate request, or it may cause the core to remain in the WAIT state, depending on the scheduler's selection and the anticipation heuristic's evaluation. In the latter case, the original timeout remains in effect, thus preventing unbounded waiting by repeatedly retriggering the timeout. The state diagram in Figure 3 illustrates this decision process.

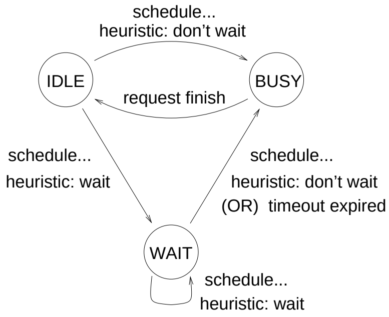


Figure 3: Waiting mechanism, state diagram

There is a scheduler-independent optimization on the above algorithm: if the process that issued the last request blocks on I/O by issuing a synchronous request, then assumption #1 suggests that a dependent request will not arrive from any other process. The anticipation heuristic can thus be short-circuited, and the chosen request immediately dispatched. This happens quite often in practice, even on occasions when the heuristic would have decided to wait further.

3.3 Seek reducing schedulers

This section describes scheduler-specific anticipation heuristics for seek reducing schedulers such as SPTF, Aged-SPTF and CSCAN. The Shortest Positioning-Time First policy [34] (also known as Shortest Time First [22] and Shortest Access-Time First [15]) calculates the positioning time for each available request from the current head position, and chooses the one with the minimum. Our goal is to design an anticipation heuristic that maximizes the expected throughput.

The heuristic needs to evaluate the candidate request chosen by the scheduling policy. The intuition is as follows: if this candidate request is located close to the current head position, then there is little point in waiting for additional requests. Otherwise, using assumption #1, if the process that issued the last request is likely to issue the next request soon (i.e., its expected median thinktime[3] is small), *and* if that request is expected to be close to the current head position, then the heuristic decides to wait for it. The waiting period is chosen as the expected 95-percentile thinktime, within which there is a 95% probability that a request will arrive.

This simple idea is generalized into a succinct cost-benefit equation, intended to handle the entire range of values for positioning times and thinktimes. Our throughput objective translates to profitably balancing the benefit of waiting, i.e., expected gains in positioning time, against the cost of waiting, which is the additional time likely to be wasted. If LP is the last request issuing process, and *elapsed* is the time passed since completion of the previous request, then:

```
$benefit = (calculate\_positioning\_time(Candidate)$
        $- \text{LP}.expected\_positioning\_time)$

$cost = \max(0,$
        $\text{LP}.expected\_median\_thinktime - elapsed)$

$waiting\_duration = \max(0,$
        $\text{LP}.expected\_95percentile\_thinktime - elapsed)$

$return\ (benefit > cost\ ?\ waiting\_duration\ :\ 0)$
```

Positioning time for the candidate request is calculated with a suitable estimator (more on this in Section 3.6). Regarding the cost estimate: for requests that arrive before the median thinktime, the heuristic expects progressively shorter periods of additional waiting; hence *elapsed* is subtracted from the expected median thinktime. However, if we wait beyond this median, the heuristic expects a request to be issued sometime very soon, and cost at this point becomes zero. Secondly, in the WAIT state, the anticipation core prevents unbounded waiting by not retriggering the timeout according to the heuristic's evaluation. Yet we calculate the correct value of waiting_duration: this is done to allow for coarse-granularity timers, so that a request arriving after the 95%ile thinktime will force an immediate dispatch as if the timeout had occurred just then.

The adaptive component of the heuristic consists of collecting online statistics on all disk read requests, to estimate the three expected times. The expected positioning time for each process is a weighted average over time of the positioning time for requests from that process, as measured upon request completion. The decay factor is set to forget 95% of the old positioning time value after ten requests, so the heuristic adapts fast. An alternate, approximate method is to track the expected *seek distance* of a request from the previous request issued by that process, and calculate expected positioning time on the fly.

Expected median and 95%ile thinktimes are estimated by maintaining a decayed frequency table of request thinktimes for each process. Thinktimes are computed from the time of completion of the last request issued by a given process, to the current time. If, however, the scheduler already has a read request queued for this same process, then this new request is treated as asynchronous and its thinktime is set to zero. The heuristic maintains 30 per-process buckets that store the count of requests that arrive after various thinktimes, ranging from 0 to 15ms at a granularity of 500μs per bucket. These bucket counts are all decayed by reducing them to 90% of their original values for every incoming request for that process. The distribution of thinktimes usually looks like a bell curve; this is consistent with assumption #2. (For many applications, the crest is located at about 1ms). The heuristic calculates the median and 95%ile points of this curve; it does all the above for every incoming synchronous request.

This heuristic is suitable for the conceptually simple SPTF policy. We now consider modifying it for two other seek reducing schedulers, namely Aged-SPTF and CSCAN.

Variant: Aged-SPTF

SPTF is known to suffer from potential starvation, since requests for distant locations on the disk may never get serviced. To bound response time, Aged-SPTF (also known as Aged-SATF and Weighted-SPTF) has been proposed as a variant: requests in the SPTF queue are associated with priorities, which are raised in some manner (often gradually) with queued time. A request with sufficiently high priority overrules the SPTF decision and gets scheduled [15, 22, 34].

The anticipation heuristic for SPTF works for Aged-SPTF also, with one minor limitation. When Aged-SPTF chooses a distant request that is too old, the SPTF heuristic would be unaware of this. It may decide to wait for additional, nearby requests. However, even if a new request from the last process arrives in this period, the scheduler then continues to pick the same old request. The last process then gets blocked, and the scheduler-chosen candidate is serviced as desired. This incurs one unnecessary thinktime on each of such occasions; this minor performance problem can be fixed by customizing the heuristic to the Aged-SPTF policy: whenever Aged-SPTF selects a request that is different from the request that SPTF would correspondingly choose, then we decide not to wait.

Variant: CSCAN: Cyclic SCAN

CSCAN (also known as C-LOOK) is an extremely popular scheduling policy, and is implemented in many Unix-based operating systems. It is the unidirectional version of Elevator/SCAN/LOOK; it moves the head in one direction, servicing all requests in its path, and then starts over at the first available request.

Our anticipation heuristic for this scheduler is based on the

[3] We define thinktime for a process issuing a request as the interval between completion of the previous request issued by the process and issue of a new request.

one for SPTF, with one additional clause. The statistics collection module in the heuristic additionally maintains a decayed expectation of the seek direction: forward or backward. On evaluating a request, if the current candidate involves a forward seek and the expected next request has a fairly high likelihood (more than 80%) of a backward seek, then we bypass the cost-benefit equation and decide not to wait. In the opposite case, we wait for the usual amount of time. For applications performing random access, with roughly 50% of the seeks pointing in each direction, this heuristic for CSCAN is not ideal. This is because CSCAN itself is poorly suited to handle this case.

3.4 Proportional-share schedulers

We next present an anticipation heuristic designed for a proportional-share scheduler like Yet-another Fair Queueing (YFQ) [7] or Stride [32]. These policies maintain weighted virtual clocks to remember the amount of disk service received by each process. A request is chosen from the process with the smallest virtual clock, so as to advance them in tandem.

Unfortunately, deceptive idleness forces these virtual clocks to go out of sync. Some processes do not generate enough requests in time, and their virtual clock lags behind. Processes that genuinely issue few disk requests also lag behind, but their expected thinktimes are high. Our heuristic is therefore as simple as waiting for the last process, if it meets three conditions: (a) it has no pending requests at the time its last requests completes, (b) it has an expected thinktime smaller than 3ms, and (c) it has a virtual clock smaller than the minimum virtual clock of processes with available requests (*minclock*). The 3ms threshold is chosen somewhat arbitrarily; there is no consistent way to balance weighted fairness against performance. 3ms is observed to be larger than the thinktimes for most applications, without being too large as to degrade performance. As before, we wait for the 95%ile point of the thinktime distribution for this process.

3.5 Heuristic combination

A proportional-share scheduler with an assignment of 1:2 can service one request from the first process for every two requests from the second. Alternatively, it can enable some seek reduction, by slightly relaxing the timescale on which it operates. This allows the scheduler to service n requests for the first process for every $2n$ requests for the second, where each set might contain sequential requests. One variation on this theme is suggested in [29], where the scheduler picks from processes with virtual clocks between *minclock* and $minclock + \tau$ (where τ is a relaxation threshold, and could be 1 second). Among these, it chooses the request with the smallest positioning time.

We propose a *combination heuristic* for this scheduler, thus hinting at general methods of combining anticipation heuristics. This combination is necessary because: (1) the heuristic for SPTF, if applied directly here, would not wait for either process if the access pattern were random, and would thus violate the proportion assignment; (2) the heuristic for Stride, if used, would wait only for the process with higher share, and thus enable only partial seek reduction.

A straightforward approach of combining these two heuristics is to separately evaluate the candidate request on each of the two, and return the larger of the two evaluations. In other words, if the waiting decision is taken for either reason, then the combination will conservatively choose to wait.

We identify and accommodate for two minor performance issues with this simplistic approach. Firstly, the Stride policy has been relaxed due to the introduction of τ. Condition (c) in the anticipation heuristic for Stride needs to be correspondingly changed from *minclock* to $minclock + \tau$.

Secondly, consider the heuristic for SPTF waiting for sequential requests from process p, and successively servicing many such requests. At some point, p's virtual clock may become larger than $minclock + \tau$, in which case the conservative decision to wait becomes pointless. Our heuristic watches for this condition and decides not to wait.

3.6 Implementation issues

There are two implementation issues that deserve elaboration, namely calculating positioning time for requests and building an inexpensive timeout mechanism.

Estimating access time for requests is nontrivial due to factors like rotational latency, track and cylinder skews, and features of modern disks like block remapping and recalibration. Nonetheless, much work has been done in this area, and it is possible to build a software-only predictor with over 90% accuracy [13, 15, 21, 35]. However, we used a much simpler logical block number based approximation to positioning time. A user-level program performs some measurements to capture the mapping between the logical block number difference between two requests and the corresponding head positioning time, and fits a smooth curve through these points. This takes about 3 minutes at disk installation time, but can be made online and non-intrusive. This method automatically accounts for seek time, average rotational latency and track buffers. It has an accuracy of about 75%, which we experimentally confirm to be sufficient, given the insensitivity of the anticipation heuristic.

There are many possible timer mechanisms to choose from. We use the i8254 Programmable Interval Timer (PIT) to generate interrupts every 500μs, and build a simple timeout system over that. Experiments demonstrate how this rather coarse-grained timer is amply sufficient for our purposes. Each interrupt causes a processing overhead of about 4μs on our hardware [2], thus causing about 1% CPU overhead on computational workloads. Other timeout mechanisms can be used in place of the i8254, if higher accuracy and lower overhead are desired. Some pentium-class processors (mostly SMPs) have an on-chip APIC that delivers fine-grained interrupts with an overhead of only 1 to 2μs per interrupt. Alternatively, soft-timers [2] pose an extremely light-weight alternative.

4. EXPERIMENTAL EVALUATION

This section evaluates the anticipatory scheduling framework on a range of microbenchmarks and real workloads. We show that this transparent kernel-level solution eliminates deceptive idleness, and achieves significant performance improvement and closer adherence to QoS objectives wherever applicable.

Code and platform: We implemented the anticipatory scheduling framework and heuristics in the FreeBSD-4.3 kernel. The code comprises of a kernel module of about 1500 lines of C code, and a small patch to the kernel for necessary hooks into the scheduler and disk driver. Unless otherwise stated, our experiments are conducted on a single 550MHz Pentium-III system, equipped with a 7200rpm IBM Deskstar 34GXP IDE disk and 128 MB of main memory.

Schedulers: All experiments with a seek reducing scheduler use Aged-SPTF unless otherwise specified. We configure this scheduler to perform shortest positioning-time first scheduling, with a bounded per-request latency of 1 second. This is found to achieve performance to within 1% of SPTF. Anticipatory scheduling involves an intrinsic latency tradeoff: servicing multiple requests from one process for seek reduction necessarily increases request turnaround time for another. However, most server-type applications would find this small increase acceptable, in exchange for significant improvements in throughput. A system that desires lower latency may reduce the above delay bound to say 100ms; this was measured to reduce the throughput by at most 8% on our system.

Metrics: Our experiments employ two metrics of application performance: the application-observed throughput in MB/s, and the disk utilization. In our framework, a disk spends time either servicing requests (i.e., positioning head and transferring data) or idling; we define *disk utilization* in an interval as the percentage of real time spent servicing requests.[4] This choice of the utilization metric depicts the fraction of time that the disk is deliberately kept idle, and helps in understanding some throughput measurements.

Turning off filesystem prefetch: Some operating systems, including FreeBSD, do not implement asynchronous prefetch in some subsystems. For example, the VM subsystem does not issue auxiliary prefetch requests for page faults that are serviced from disk. Similarly, FreeBSD also does not perform asynchronous prefetch for `sendfile()` and `readdir()`. This allows us to effectively turn off prefetching for evaluation purposes, by mapping files to memory and accessing the memory locations.

Two sets of microbenchmarks, exhibiting variations in access patterns and thinktimes, serve to illuminate the workings of anticipatory scheduling as applied to seek reducing schedulers.

4.1 Microbenchmark: Access patterns

We study the effect of anticipatory scheduling on synchronous requests issued in different access patterns, with and without filesystem prefetch enabled. Two processes rapidly issue 64 KB disk read requests into separate 1 GB files; these are either sequential (*seq*), or target every alternate 64 KB chunk (*alter*), or are randomly positioned within their respective files (*random*). Some experiments use the `read` system call, for which FreeBSD 4.3 transparently issues asynchronous prefetch requests if the access pattern is detected to be sequential on disk. Other experiments map their file into memory using `mmap`, and fault on the memory pages; these are not subject to asynchronous prefetch. Figure 4 shows the results.

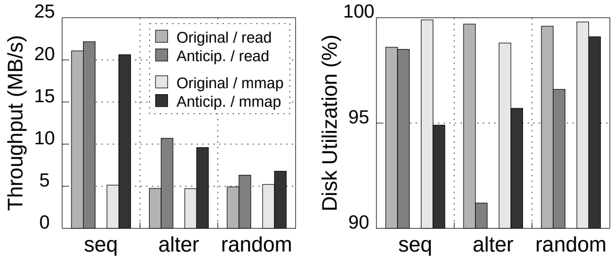


Figure 4: Impact of anticipatory scheduling on disk throughput and utilization, using sequential, alternate-block and random access workloads, and read versus mmap based access.

Asynchronous prefetch ensures that sequential accesses using read achieve almost full disk bandwidth (about 21 MB/s). However, filesystems often lay out logically contiguous blocks of a large file as a set of separate regions on disk. On the infrequent occasions that a boundary is crossed, FreeBSD's prefetching mechanism temporarily assumes non-sequential access and conservatively backs off. Anticipatory scheduling waits for such processes, thus exploiting spatial locality within the large file. Performance improves by about 5%, by steadily fetching blocks from one file until Aged-SPTF forces it to switch.

Since mmap-ed accesses are not subject to prefetch, anticipatory scheduling attains four times better throughput than the original case. This achieves throughput almost equal to the maximum disk bandwidth; the 6% difference between the two is reflected by an almost equal fraction of time that the disk is kept idle. This `mmap` case is arguably a shortcoming of FreeBSD's prefetch implementation. However, as exemplified in the following two cases of *alter* and *random*, non-sequential disk access using `read` can use anticipatory scheduling to significantly improve throughput wherever prefetching fails.

Consider the second set of experiments, where alternate blocks are read. This defeats the FreeBSD prefetch heuristic, causing both read and mmap to achieve only 5 MB/s. Anticipatory scheduling improves throughput to the maximum that can be achieved for alternate blocks, i.e., half the disk bandwidth. We will see several variants of such non-sequential access in real workloads.

Lastly, in the random access case, the smaller improvements (28% and 30%) by anticipatory scheduling are because each process is performing random access within its respective file, so gains are mostly due to seek reduction between files.

4.2 Microbenchmark: Varying thinktimes

The next set of four microbenchmarks illustrates the impact of waiting on applications that take different amounts of time to issue the next request. Two processes map separate, large files into memory, and fault on these memory pages sequentially (thus without asynchronous prefetch). After every 64 KB, they pause for some amount of time as described below.

[4] In contrast, a work-conserving scheduler never idles for a busy workload, and might prefer to define utilization as the percentage of service time spent transferring data from disk.

4.2.1 Symmetric processes:

Consider Figure 5, where time t on the horizontal axis represents the duration in milliseconds that *each* process spends waiting between requests. Each data point in the throughput graph is a separate experiment. For values of t up to 8ms, the original system alternates between requests from the two processes, achieving only 5 MB/s. When thinktime exceeds 8ms, the waiting time becomes comparable to request service time, and utilization for the original system starts falling below 100%. Occasionally, deceptive idleness is avoided by servicing two successive requests for the same process. This fades away for larger values of t.

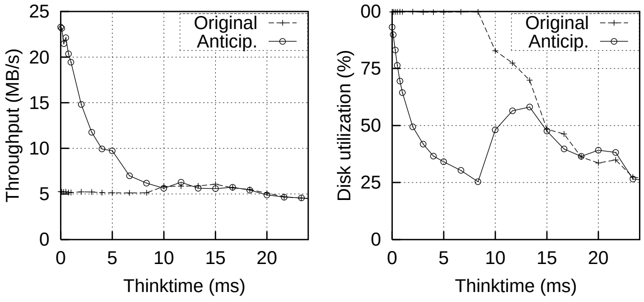


Figure 5: Increasing thinktimes for both processes

With anticipatory scheduling enabled, the situation changes as follows: When $t = 0$, we see the familiar situation where throughput is four times that of the original system. For larger values of t up to 8ms the effect of waiting becomes increasingly burdensome on throughput and utilization, and the improvement steadily declines. At about 8ms, the waiting time becomes comparable to request service time, and the cost-benefit equation tips the other way. Performance then approaches that of the original system to an increasing degree. Measurements indicate that many applications have very short thinktimes when busy, in the region of 200μs to 2ms. Hence, anticipatory scheduling is expected to achieve significant benefits on real applications.

4.2.2 Asymmetric processes:

Consider an alternative scenario in Figure 6 where only one (slow) process waits for duration t between requests, while the other (quick) process issues request as soon as its previous request completes. The original system alternates between the two processes' requests for t up to 12ms, but beyond that, two or more requests arrive from the quick process for every request from the slow one. This causes partial avoidance of deceptive idleness, due to which performance gradually improves for increasing t.

With anticipatory scheduling enabled, the attained throughput exceeds that of the original system by a large margin. The anticipation heuristic is greedy, and for small values of thinktime, it decides to wait for both processes. This results in a gradual throughput decrease with increasing thinktime, until a point is reached (4ms) where the heuristic waits for the quick process but not for the slow process. Throughput rises back to the maximum, with requests from the slow process serviced only when Aged-SPTF induces a switch. Note that Aged-SPTF only guarantees non-starvation, not fine-grained fairness.

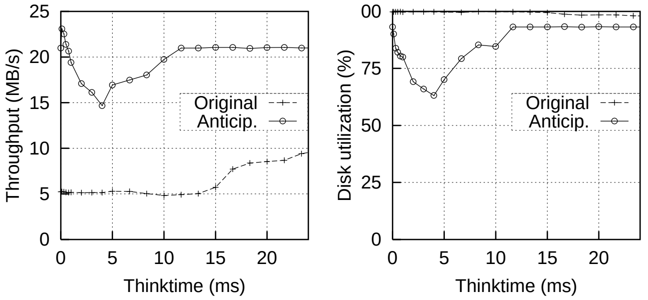


Figure 6: Increasing thinktimes for one process

4.2.3 Random thinktimes:

Next, we seek to understand how well the anticipation heuristic adapts to thinktimes that vary rapidly within an experiment. Interestingly, if a process waits for a random duration uniformly distributed between 0 and t, it performs almost as well as the deterministic counterpart. This is because the expected median thinktime is judged to be roughly $t/2$, and the expected 95%ile thinktime becomes almost t.

4.2.4 Adversary:

Since the heuristic copes with randomly varying thinktimes, we try to exercise the pathological-case behaviour of the heuristic by writing an intelligent adversary. Two symmetric processes wait for a duration determined as follows: they issue n rapid requests, then wait for a duration that just exceeds the timeout set by the heuristic, and repeat. This application actively fails to comply with assumption #2, and thus encumbers the heuristic from adapting effectively. Results for varying n are shown in Figure 7. For $n = 0$, the anticipatory scheduler can cope with all requests arriving slowly. But for n between 1 and 4, the anticipation heuristic performs only slightly worse than the original system: by about 20%. This result indicates that even for a malicious application, or when the assumptions in Section 3.1 do not hold, the possible performance degradation is acceptably small.

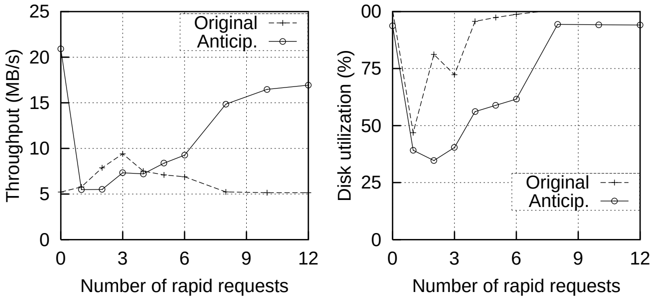


Figure 7: Adversary application

The adversary issues several requests rapidly, followed by a long wait. Interestingly, a similar situation arises in practice when applications issue very large read requests (say 1 MB), and the FreeBSD kernel breaks them up into 128 KB chunks. In this case, the scheduler receives eight 128 KB requests in rapid succession, followed by the application's typically larger thinktime period. We solve this special case by having the filesystem flag such requests, whereupon the anticipatory scheduling core treats them like one large request.

The adversary application causes many timeouts to expire, and thus stresses the accuracy of the timer. In order to understand the sensitivity of our results to the timer frequency, we reran the experiment with timer granularities of 50μs, 200μs, 500μs, and 1ms. Although the throughput peaked at 500μs (because larger timeouts allow for the occasional heuristic error), the greatest difference we saw among the four trials was only 10%. This was also supported by a similar experiment with the Apache webserver, where the difference was negligible.

Solving deceptive idleness can clearly bring about significant benefits on microbenchmarks, but what is its impact on real applications? To see this, we use two real applications (webserver and linker), and two standard benchmarks (filesystem and database) that are expected to reflect a wide range of application workloads.

4.3 The Andrew filesystem benchmark

The Andrew Benchmark [12] attempts to capture a typical fileserver workload in a software development environment. It consists of k clients, each performing five phases: (a) *mkdir*, which creates n directories, (b) *cp*, which copies a standard set of 71 C source files to each of these n directories, (c) *stat*, which aggressively lists all directory contents, (d) *scan*, which reads all these files using `grep` and `wc`, and finally (e) *gcc*, which compiles and links them. We configured n to be 500, so that the repository size exceeds main memory. We call this set of n directories a *repository*, and instantiate one such repository for each of the $k = 2$ clients, aiming to simulate concurrent access to a fileserver. This experiment uses the same Aged-SPTF scheduler as before, with and without anticipatory scheduling enabled.

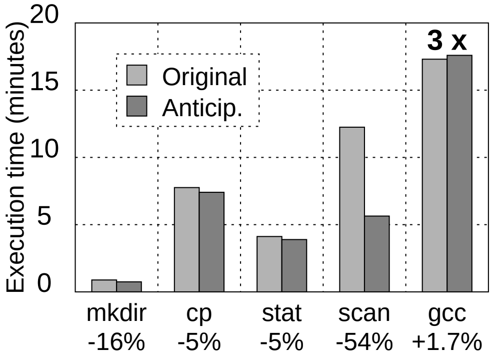


Figure 8: The Andrew Benchmark. The last pair of bars are shown scaled down by a factor of 3.

A breakup of the execution times for individual benchmark phases is presented in Figure 8. Consider the scan phase, which is the only one that issues streams of synchronous read requests. Anticipatory scheduling transparently reduces execution time for this phase by 54%. Both `grep` and `wc` on FreeBSD use `read`, not `mmap`, and would thus benefit from kernel prefetch. However, individual files are small, so this prefetch has little effect. Major seek reduction happens here due to the files being in the same directory, and thus closely positioned on disk. Anticipatory scheduling enables the scheduler to capitalize on these seek opportunities and halve the execution time.

Other disk-intensive phases improve by smaller amounts: 16% for mkdir with metadata writes, and 5% for cp and stat each (the latter typically gets cached in memory). The gcc phase is CPU-bound, but also performs some disk I/O; this aptly demonstrates the overhead of our system. There is an increase in execution time by 1.7%, due to two factors: CPU processing for the additional i8254 timer interrupts, and the CPU overhead corresponding to the heuristic execution routines (mainly statistics collection). This phase strongly dominates total execution time, so that the overall benchmark shows the smaller improvement of 8.4%.

Performance with one client is the same with or without anticipatory scheduling; indeed, when there is only one stream of synchronous requests, anticipatory scheduling plays no role. Increasing the number of clients from 2 to 8 shows almost no performance difference: the scan phase improves by 57% in the latter case. This confirms the applicability and scalability of anticipatory scheduling to busy fileservers.

4.4 The Apache webserver

The Apache webserver employs a multi-process architecture to service requests from clients. Requests that miss in the main-memory cache are serviced from disk by the respective process. This happens frequently for webservers with large working sets, to the point of becoming disk-bound. In its default configuration, Apache-1.3.12 (and also 2.0a9) mmaps files that are smaller than 4 MB, and writes it out to a network socket. For larger files, Apache reads the data into application buffers first; this was done to prevent a swap-based DoS attack on IRIX systems. Many other webservers and ftp servers use similar mechanisms for file transfer.

We first configure Apache to exclusively use either read or mmap in a given experiment. We run the Apache webserver with 3 client machines which host 16 client processes each. Real websites have different amounts of concurrency, depending on amount and characteristics of incident load; we therefore varied the number of clients over a wide range, and observed very little difference in results. These clients rapidly play requests from a trace selected from the CS department webserver at the University of California, Berkeley [6]. These requests have a median size of 4768 bytes, a mean size of 86 KB, and a mean size of 13 KB if the largest 5% of the requests are excluded. This trace is quite disk-intensive, so 1000 requests target 745 distinct files. The scheduler, as before, is Aged-SPTF.

Figure 9 characterizes the observed throughputs and utilizations. We observe a 29% improvement in throughput for read, where anticipatory scheduling complements filesystem prefetch, and a larger 71% improvement for mmap (without prefetch). Unlike in the Andrew Benchmark, all Apache clients generate requests to the same repository, so requests to an individual Apache process do not exhibit much locality *across files*. So seek reduction opportunities are mainly in terms of servicing each file fully before moving on to the next. Many files are too small for any seek reduction. *Intermediate-sized files* are potential candidates for prefetching, but filesystem prefetch is conservative and does not occur until a threshold number of requests are found to

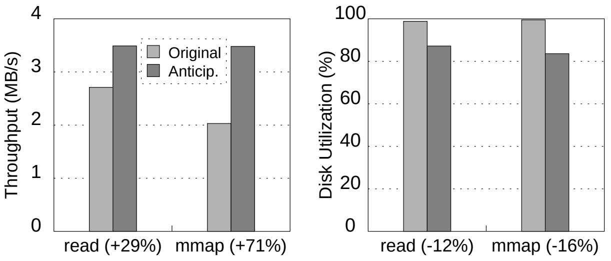


Figure 9: The Apache Webserver configured in two modes, read and mmap. The former exemplifies the practical limitations of filesystem prefetch.

be sequential. Anticipatory scheduling effects the 29% improvement in this domain. Prefetch occurs for reads on large files, but not for mmap. This accounts for the large difference in performance between the two methods of access. In the default configuration (with mmap or read depending on file size), Apache yields 2.2 MB/s on the original system and 3.5 MB/s with anticipatory scheduling; this improvement of 59% lies between those for the read and mmap cases.

4.5 The GnuLD linker

This experiment involves the last stage of a FreeBSD kernel build, starting from a cold filesystem cache. The GNU linker reads 385 object files from disk. 75% of these files are under 10 KB, whereas 96% are under 25 KB. After reading all their ELF headers, GnuLD performs up to 9 (but usually about 6) small, non-sequential reads in each file, corresponding to each ELF section. These reads are separated by computation required for the linking process.

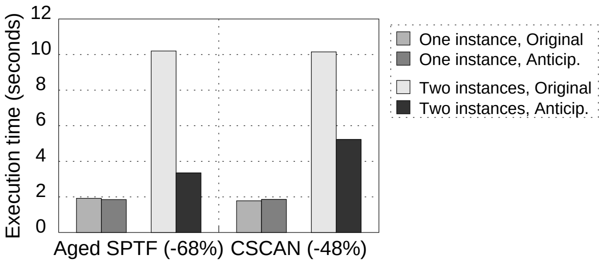


Figure 10: The GNU Linker: multiple, concurrent instances cause deceptive idleness, which is eliminated by anticipatory scheduling.

The experiment in Figure 10 demonstrates the performance of one and two simultaneous instances of GnuLD on disjoint repositories. We use two schedulers this time, Aged-SPTF and CSCAN, to demonstrate the impact of their respective heuristics. With one synchronous request issuer process, both schedulers result in execution times of about 1.8 seconds each. We would normally expect this to double for two instances of GnuLD. However, deceptive idleness causes an increase in execution time by a factor of 5.5 instead. This is again because non-sequential accesses preclude transparent filesystem prefetching.

Anticipatory scheduling brings about a benefit of 68% in the Aged-SPTF case, and causes performance to scale almost exactly as expected (i.e., to twice the execution time of a single process). The CSCAN scheduler, on the other hand, always services requests in the forward direction. But object files are accessed in arbitrary order; CSCAN therefore intrinsically precludes anticipatory scheduling from attaining the full potential for seek reduction. We see a performance improvement of only 48%; this execution time is 56% higher than the Aged-SPTF case.

4.6 The TPC-B database benchmark

The TPC-B benchmark, specified by the Transaction Processing Council in 1994, exercises a database system on simple, random, update-intensive operations into a large database, and is intended to reflect typical bank transactions [27]. Though it is considered outdated, it serves to illustrate the impact of anticipatory scheduling on a read-write workload.

We implement the above with a MySQL database and two client processes. However, we somewhat deviate from the setup specified in TPC-B; our main goal is to demonstrate the gains due to anticipatory scheduling, rather than to obtain performance data for our hardware configuration. (1) Individual records in the database are required to be *at least* 100 bytes large. MySQL has computational overheads that made it CPU-bound for record sizes of 100 bytes, so we use 4 KB records to make data I/O the bottleneck. (2) We use a database size of 780 MB, thus considerably exceeding the 128 MB main memory size; our hardware is capable of supporting larger databases. (3) MySQL does not support transactions. Many databases maintain a transaction log, which could potentially become the performance bottleneck. (4) Figure 11 depicts four experiments. The clients in the first two experiments issue `update` queries as required by TPC-B, but those in the last two replace the update operation by a `select`. (5) Finally, both clients in the first and third experiments issue queries directed at the same database, as required by TPC-B. The second and fourth experiments are a variant, where the two clients issue requests to *two separate databases*.

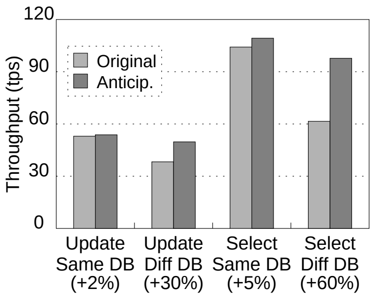


Figure 11: The TPC-B database benchmark and variants: two clients issuing update versus select queries into the same versus different databases.

An update query reads the record first, and then issues an asynchronous delayed write request. The presence of enough delayed writes can give the scheduler more choices, and alleviate the effect of deceptive idleness. Also, seek reduction within a database is severely limited due to almost random queries therein, so the first experiment shows a net improvement of only 2%. The second experiment physically separates the two databases on disk; the impact of anticipatory scheduling is now more pronounced due to seek reduction opportunities within and between databases, and we observe a 30% improvement despite the delayed write requests. Absolute performance is understandably lower than in the first case, due to large seeks between the two databases. Finally, gains due to anticipatory scheduling are best brought out in the absence of any delayed writes, i.e., when the update operation is reduced to just a select, involving one synchronous read request. We observe throughput improvements by 5% and 60% for requests to the same and different databases respectively.

In summary, our experiments indicate that a database-like workload often stands to gain by the transparent deployment of anticipatory scheduling in the operating system. However, modern commercial databases are highly optimized, and it is likely that they implement some form of application-level prefetching; we have not explored this issue further.

4.7 Proportional-share Scheduling

This experiment demonstrates the impact of the anticipation heuristic for proportional-share schedulers, and the combination heuristic. The workload is chosen to be the fourth TPC-B variant in the database experiment above: `select` operations on different databases, to achieve throughputs of 61 and 98 transactions/sec (i.e., 60% improvement with anticipatory scheduling).

Figure 12 depicts an experiment where this workload is subject to proportional scheduling. We use the Stride scheduler augmented with underlying seek reduction, as described in Section 3.5; the relaxation threshold τ is set to 1 second. Proportions of 1:2 are assigned to the two TPC-B clients p and q; these are in terms of disk utilization (not throughput, without loss of generality). In the three cases, the anticipation framework is either disabled, or separately configured with the Stride or the combination heuristic respectively.

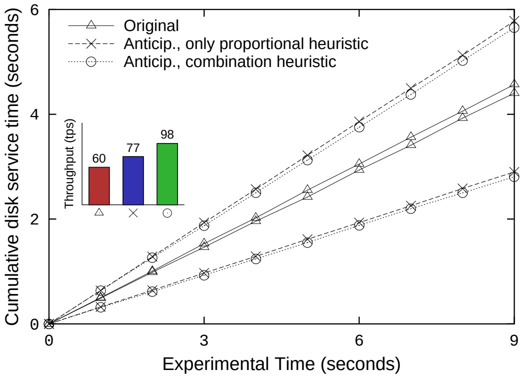


Figure 12: Proportional-share scheduler. Three experiments: (△) original: 1:1 proportions, (×) anticipatory with proportional heuristic: 1:2 proportions, and (○) anticipatory with combination heuristic: 1:2 proportions with maximum throughput.

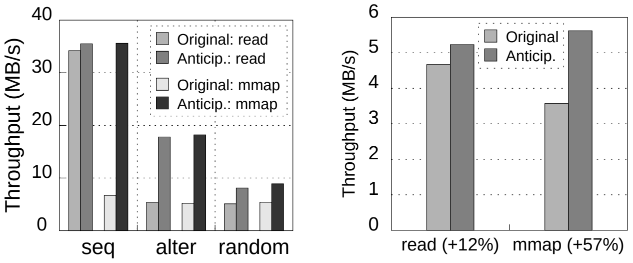
In the original system, the scheduler always multiplexes between requests from the two processes, and incorrectly achieves proportions of approximately 1:1, with the fairly low throughput of 60 tps. When we turn on anticipatory scheduling with the heuristic for proportional-share schedulers, it realizes that process q (with the higher share) is lagging behind, and waits for it. With average seek and transfer times of 9ms and 3ms, the scheduler manages to achieve 1:2 proportions by servicing 5 requests from q for every request from p. This is sufficient to exploit locality between requests of *one process*, namely q; throughput improves to 77 tps, i.e., by about half the maximum possible. This results in a corresponding total utilization drop of about 2%, as is seen by utilizations of both processes decreasing proportionally.

The combination heuristic, on the other hand, realizes the seek reduction potential in waiting for both processes. It thus services several requests from each process, and achieves the full 98 tps throughput, while retaining proportions of 1:2.

4.8 Advanced hardware

We wish to determine the effect of anticipatory scheduling on modern hardware, using the next generation CPUs, disks and controllers. Studies indicate that head seek time improves more slowly than data transfer time; this trend will further aggravate the effects of deceptive idleness. Functionality supported by modern controllers like tagged queueing and improved track buffering and controller-level prefetching may become underused for synchronous I/O. On the other hand, track buffering may assist filesystem prefetching for medium-sized sequentially accessed files, and thus alleviate the problem in some cases. Track buffering also allows the scheduler to wait for the next request, without requiring a complete rotation to read the adjacent sector. On a different note, an increase in CPU speed corresponds to a reduction in application thinktime, which is advantageous for waiting. Thus, a number of tradeoffs can influence the precise gains due to anticipatory scheduling.

To explore this issue, we perform some experiments on an 800MHz Athlon system, with a 15,000 rpm Seagate Cheetah ST318451LW SCSI-3 disk and an Adaptec 19160B Ultra160 controller. Specifically, we repeat two experiments: the microbenchmark with different access patterns (Section 4.1) and the Apache webserver experiment (Section 4.4). Results are in Figure 13.



(a) Microbenchmark

(b) Apache webserver

Figure 13: Experiments performed on advanced hardware: 15,000 rpm SCSI disk, 800 MHz CPU.

We note that the maximum bandwidth on this disk is 55% higher than on our original IDE disk, due to a corresponding increase in rotational speed. However, deceptive idleness causes both disks to deliver nearly the same low throughput in the presence of large seeks; this magnifies the best-case gains of anticipatory scheduling to a factor of 5.5, as compared to the earlier factor of 4. Other aspects of this microbenchmark are similar to those on the IDE disk.

Next, consider the Apache webserver experiment. Improvements for the `read` and `mmap` configurations are 12% and 57%. While this is still significant, it is lower than the IDE counterparts. Improved rotational speed, different disk geometry and better track buffering result in relatively faster servicing of short seeks; these are common in the Apache workload, thus leading to smaller improvements.

To summarize, modern hardware does suffer from deceptive idleness, and stands to gain from anticipatory scheduling. The actual improvements expected on future hardware can be either more or less, depending on precise hardware details and application characteristics.

On a related note, we consider the impact of deceptive idleness and anticipatory scheduling on other disk types, such as redundant arrays of inexpensive disks (RAIDs), just a bunch of disks (JBODs), and network disks. We have not investigated this issue in sufficient depth, but we believe that deceptive idleness can affect such disks, and that anticipatory scheduling can be beneficial. The positioning time estimator would need to derive a useful model of device behaviour, including head positions and redundant copies of data; we believe that this is the key step to adapting anticipatory scheduling to such hardware.

5. DISCUSSION

This section discusses the practical impact of anticipatory scheduling, and suggests improvements to its design.

5.1 Relevance of anticipatory scheduling

Many applications perform non-sequential read I/O on large files, or access many small files colocated on disk, such as those in the same directory. Applications such as webservers and databases often have huge working sets, and issue read requests that cannot be satisfied from memory. This general tendency of applications to issue concurrent, synchronous, non-sequential disk requests has been on the rise [19, 30]. These requests typically do not benefit from traditional filesystem prefetching, and yet possess enough locality to be excellent candidates for seek reduction. This has driven the need for an alternative and more general approach to complement prefetching. Since anticipatory scheduling is based on a much weaker form of prediction, it is feasible in many situations where prefetching is difficult.

Proportional-share schedulers are increasingly gaining prominence in modern systems; for example, they are used in various high-level quality of service systems like using reservation domains to isolate co-hosted websites [8], and performing admission control to guarantee predictable performance of webservers [3]. It is important for these disk schedulers to adhere to their contract; anticipatory scheduling facilitates this for applications issuing synchronous I/O. In practice, proportional-share disk schedulers will almost always be deployed in combination with a seek reducing scheduler [29]. Our experiments have demonstrated how the combination heuristic brings about simultaneous improvement of both contract adherence and performance.

Real-time disk schedulers (either pure or in combination with seek reducing schedulers) are commonly used to serve and view multimedia content [9, 11]. Under certain circumstances, it is possible for deceptive idleness to cause such schedulers to multiplex between requests from different processes, and consistently violate deadlines. We believe that the anticipatory scheduling framework is applicable to real-time scheduling, but a full exploration of the design and merits of an anticipation heuristic is beyond the scope of this paper.

5.2 Potential improvements

We suggest two approaches to improve on our proposed design. These are aside from the obvious improvements of making the timing mechanism and the positioning time estimator cheaper and more accurate.

5.2.1 *Accumulate more statistics*

It is possible for the anticipation heuristic to make suboptimal decisions. We can reduce this chance by augmenting its adaptation mechanism with additional statistics:

(1) Besides tracking expected thinktimes and positioning times, we could collect statistics about the *variance* of these estimates. This gives the heuristic an idea of how accurate these estimates really are. We could then use a technique such as *covariance resetting* to discard all previously accumulated statistics whenever this variance becomes too high. (2) The heuristic could keep track of how frequently timeouts expire for each process; if this exceeds some threshold rate, then regardless of all other notions of accuracy, it would know that something is wrong. (3) The positioning time estimator may not be accurate; however, it can measure positioning time after a request has completed service. This provides an indicator for the error in estimation, and thus, our confidence in future decisions. (4) An application might use `aio_read` to issue requests that are actually synchronous; the heuristic can determine this post-facto, and remember it to optimize future decisions.

5.2.2 *Relax the two workload assumptions*

The anticipatory scheduling framework waits for the last request issuing process, and collects statistics at a process granularity. Though this is easily the common case, relaxing the assumptions in Section 3.1 can enable the anticipation heuristics to support a wider range of applications, of the following types:

(1) Some proportional-share disk schedulers have a notion of resource principals different from processes, like resource containers [4] and reservation domains [8]. (2) Also, sometimes a group of processes may collectively issue synchronous requests. (3) Applications may simultaneously generate different access patterns on different file descriptors. (4) Some programs may issue two kinds of disk requests from two different parts of the program code, but on the same file descriptor. (5) Seek reduction intrinsically deals with requests in the same region on the disk; online clustering can classify requests into groups.

To relax the assumptions, the heuristic can collect statistics at all levels of abstraction, i.e., processes, threads, instruction pointer for thread, file descriptors, and disk region –

along with their variances. The heuristic can then choose the *highest consistent level* out of these. This has low variance, is expected to be correct, and contains most information.

6. RELATED WORK

This section points out interesting phenomena analogous to deceptive idleness, and methods related to anticipatory scheduling, in each of three domains: disk, CPU and network interface scheduling.

Anticipatory scheduling is based on the non-work-conserving scheduling discipline. To our knowledge, the only other non-work-conserving disk scheduler solves a memory management issue for mixed real-time and best-effort workloads. It refrains from servicing all outstanding best-effort requests, and conserves buffer space for future real-time requests [10].

The basic idea of anticipatory disk scheduling has been independently suggested in a posting to the Linux-kernel mailing list – coincidentally under the same name [25].

For write requests, the AIX operating system implements *I/O pacing* to prevent programs from saturating the system's I/O facilities. This enforces per-file high and low water marks on the number of queued requests [33]. This low water mark buffers write requests and increases opportunities for seek reduction; it can be viewed as the counterpart of anticipatory scheduling for delayed write requests.

Also in the context of efficiently handling asynchronous requests, *freeblock scheduling* [16] has been proposed to increase media bandwidth utilization by potentially servicing asynchronous requests enroute to the synchronous ones.

Filesystem prefetching is a well-researched area [24], and for regular workloads, asynchronous prefetch can transparently eliminate deceptive idleness (Section 2.1). There is a large body of work in improving the feasibility and effectiveness of prefetch using techniques such as application-level hints [18] and transparent compiler-directed approaches [17].

Deceptive idleness creates a momentary shortage of suitable requests; a different type of scheduler starvation arises in the context of the Aged-SPTF scheduler. Recall from Section 3.3 that priorities are assigned to requests in the SPTF queue, and these are increased over time. If this increase is performed abruptly at some time threshold, and if the rate of incoming requests exceeds service rate, then every request choice will get forced, and the scheduler degenerates to FCFS. The solution in this case involves gradually raising request priorities [15].

The CPU scheduling discipline being preemptible, there is no analog of deceptive idleness. There is, however, the equivalent of high preemption cost in switching between processes: *affinity scheduling* attempts to schedule between many threads to improve cache reuse [28]. On a different note, non-work-conserving CPU schedulers have been motivated by the need to handle bursty and unexpected workloads; these are based on maintaining one or more CPUs in reserve [20]. Similarly, non-work-conserving request schedulers have been used to support prioritized workloads in web content hosting, for differentiated levels of service [1]. In comparison, anticipatory disk scheduling is a distinctly different type of non-work-conserving scheduling.

The network *packet scheduling* discipline is non-preemptible, but deceptive idleness is unlikely in this domain. High bandwidth-delay products drive applications to maintain windows of outstanding requests, due to which the packet scheduler never faces a shortage of requests from an individual flow. Interestingly, there is reason to optimize in the opposite direction: context switching overhead is negligible, and it is important to avoid burstiness. $\mathrm{WF}^2\mathrm{Q}$ is a work-conserving scheduling policy that tries to interleave requests as much as possible, more than even WFQ does [5]. Finally, non-work-conserving schedulers have been used in packet scheduling by Zhang and Knightly to handle bursty workloads, by holding packets in the network and simulating the original traffic stream [36].

7. CONCLUSION

This paper identifies the problem of *deceptive idleness* in the disk subsystem, and proposes the *anticipatory scheduling framework* as a general and effective solution. This simple, application-transparent method brings about significant improvements in throughput and adherence to quality of service objectives for synchronous disk I/O. The framework consists of a scheduler-independent core, with separate anticipation heuristics proposed for a variety of seek reducing and proportional-share schedulers to address their disparate needs. This solution complements prefetching techniques deployed at the application and kernel levels, and is most useful in frequently occurring situations where prefetching is difficult or infeasible. It is easy to implement, and suited for incorporation into general-purpose operating systems.

This paper evaluates anticipatory scheduling under a range of workloads. Microbenchmarks characterize the intrinsic properties of the solution, whereas real applications and standard benchmarks evaluate its applicability and effectiveness in realistic scenarios. The Apache webserver is found to deliver 29% and 71% more throughput in two configurations. The Andrew filesystem benchmark runs faster by 8% (54% for the synchronous phase). Variants of the TPC-B database benchmark exhibit improvements between 2% and 60%. Proportional-share schedulers become empowered to deliver application-desired proportions for synchronously issued requests. All this is accomplished with little overhead.

8. ACKNOWLEDGMENTS

We are grateful to Margo Seltzer (our shepherd) and our anonymous reviewers for their detailed feedback, and to Juan Navarro, Karthick Rajamani and Arvind Sankar for several insightful discussions. This work was supported in part by NSF Grant CCR-9803673, by Texas TATP Grant 003604, by an IBM Partnership Award, and by an equipment donation from HP Labs. We thank the Massachusetts Institute of Technology for its hospitality during our visit in Spring 2001.

9. REFERENCES

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated quality of service in web hosting services. In *WISP*, June 1998.

[2] M. Aron and P. Druschel. Soft timers: Efficient microsecond software timer support for network processing. In *17th ACM SOSP*, Dec. 1999.

[3] M. Aron, S. Iyer, and P. Druschel. A resource management framework for predictable quality of service in web servers, July 2001. Submitted. http://www.cs.rice.edu/~ssiyer/r/mbqos/.

[4] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *3rd USENIX OSDI*, Feb. 1999.

[5] J. Bennett and H. Zhang. WF^{2}Q: Worst-case fair weighted fair queueing. In *IEEE Infocom*, Mar. 1996.

[6] HTTP log files at the University of California, Berkeley. http://www.cs.berkeley.edu/logs/http/.

[7] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE ICMCS*, June 1999.

[8] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. In *USENIX 1998 Annual Technical Conference*, June 1998.

[9] S. Chen, J. A. Stankovic, J. F. Kurose, and D. Towsley. Performance evaluation of two new disk scheduling algorithms for real-time systems. *Journal of Real-Time Systems*, 3(3):307–336, Sept. 1991.

[10] L. Golubchik, J. C. S. Lui, E. de Souza e Silva, and H. R. Gail. Evaluation of tradeoffs in resource management techniques for multimedia storage servers. In *IEEE ICMCS*, June 1999.

[11] P. Goyal, X. Guo, and H. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *2nd USENIX OSDI*, Oct. 1996.

[12] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[13] L. Huang and T. Chiueh. Implementation of a rotation latency sensitive disk scheduler. Technical Report ECSL-TR81, SUNY, Stony Brook, Mar. 2000.

[14] S. Iyer and P. Druschel. The effect of deceptive idleness on disk schedulers. Technical Report CSTR01-379, Rice University, June 2001.

[15] D. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7rev1, Hewlett-Packard, Feb. 1991.

[16] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *4th USENIX OSDI*, Oct. 2000.

[17] T. Mowry, A. Demke, and O. Krieger. Automatic compiler inserted I/O prefetching for out-of-core applications. In *2nd USENIX OSDI*, Oct. 1996.

[18] H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *15th ACM SOSP*, Dec. 1995.

[19] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *USENIX Annual Technical Conference*, June 2000.

[20] E. Rosti, E. Smirni, G. Serazzi, and L. W. Dowdy. Analysis of non-work-conserving processor partitioning policies. *Lecture Notes in Computer Science*, 949:165–181, 1995.

[21] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[22] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *USENIX Winter Technical Conference*, Jan. 1990.

[23] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. In *ACM Sigmetrics*, June 1998.

[24] E. Shriver, C. Small, and K. Smith. Why does file system prefetching work? In *USENIX Annual Technical Conference*, June 1999.

[25] J. B. Siegal, Jan. 2000. http://www.cs.rice.edu/~ssiyer/r/antsched/linux.html.

[26] D. Sullivan and M. Seltzer. Isolation with flexibility: A resource management framework for central servers. In *USENIX Annual Technical Conference*, June 2000.

[27] Transaction Processing Performance Council. TPC-B standard specification, revision 2.0, 1994.

[28] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for shared memory multiprocessors. In *13th ACM SOSP*, Oct. 1991.

[29] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared memory multiprocessors. In *ASPLOS*, Oct. 1998.

[30] W. Vogels. File system usage in Windows NT 4.0. In *17th ACM SOSP*, June 2000.

[31] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *1st USENIX OSDI*, Nov. 1994.

[32] C. Waldspurger and W. Weihl. Stride scheduling: Deterministic proportional resource management. Technical report, MIT/LCS/TM-528, June 1995.

[33] F. Waters. *AIX performance tuning guide, chapter 8.* Prentice Hall, 1994.

[34] B. Worthington, G. Ganger, and Y. Patt. Scheduling algorithms for modern disk drives. In *ACM Sigmetrics*, 1994.

[35] X. Yu, B. Gum, Y. Chen, R. Wang, K. Li, A. Krishnamurthy, and T. Anderson. Trading capacity for performance in a disk array. In *4th USENIX OSDI*, Oct. 2000.

[36] H. Zhang. Providing end-to-end performance guarantees using non-work-conserving disciplines. *Computer Communications*, 18(10), Oct. 1995.

http://www.cs.rice.edu/~ssiyer/r/antsched/