Adaptive Disk Scheduling with Workload-dependent Anticipation Intervals *

Konstantinos Tsakalozos, Vassilis Stoumpos, Kostas Saidis and Alex Delis
University of Athens, Athens 15784, Greece
{k.tsakalozos, stoumpos, saiko, ad}@di.uoa.gr

July 12, 2008

Abstract

Anticipatory Scheduling (*AS*) of I/O requests has become a viable choice for block-device schedulers in open-source OS-kernels as prior work has established its superiority over traditional disk-scheduling policies. An *AS*-scheduler selectively stalls the block-device right after servicing a request in hope that a new request for a nearby sector will be soon posted. Clearly, this decision may introduce delays if the anticipated I/O does not arrive on time. In this paper, we build on the success of the *AS* and propose an approach that minimizes the overhead of unsuccessful anticipations. Our suggested approach termed *Workload-dependent Anticipation Scheduling* (*WAS*), determines the length of every anticipation period in an on-line fashion in order to reduce penalties by taking into account the evolving spatio-temporal characteristics of running processes as well as properties of the underlying computing system. We harvest the spatio-temporal features of individual processes and employ a system-wide process classification scheme that is re-calibrated on the fly. The resulting classification enables the disk scheduler to make informed decisions and vary the anticipation interval accordingly, on a per-process basis. We have implemented and incorporated *WAS* into the current *Linux* kernel. Through experimentation with a wide range of diverse workloads, we demonstrate *WAS* benefits and establish reduction of penalties over other *AS*-scheduler implementations.

Indexing Terms: Scheduling of block-devices, anticipatory scheduling, kernel structures.

*This work was partially supported by a European Social Funds and National Resources Pythagoras grant with No.7410 and the Univ. of Athens Research Foundation.

1 Introduction

Effective disk scheduling remains a key aspect in contemporary computing systems as it significantly influences the behavior of the kernel and helps better utilize block-devices. As improvements on both CPU-speed and access to main-memory continue to out-pace advances in disk technology along with application-footprints growing larger and more demanding in terms of I/Os, the scheduling of block-devices remains a critical issue in the overall performance of the OS kernel [Lov05, VC90, WGP94, SGG03]. Traditional block-device schedulers such as Shortest-Seek-latency-First (SSF), SCAN, Look and variants [Fin86, SV06, Tan01] mainly target single disk systems. These schedulers attempt to minimize the distance traveled by the disk arm, since arm movements are orders of magnitude more time-consuming than the time spent to actually transfer data between the block medium and the main-memory. The presumed operational pattern in the above work-conserving schedulers is straightforward: as soon as a process gets its request serviced it is considered "idle" in terms of I/O activity. To this end, the scheduler serves a request posted by possibly another process. However, this "idleness" is only deceptive as the just-serviced process was actually allowed no CPU time to post a new request.

The *Anticipatory Scheduling* (*AS*) takes a different approach in deciding what is the next block request to be served. Instead of immediately dispatching to the next queued I/O –issued very likely by another process– the scheduler stalls the device for a short time period. Such a pause offers the opportunity to the process-just-serviced to post a new I/O for a nearby sector which might be attended ahead of other already awaiting I/Os in the queue of the block-device. This in effect allows *AS* to overcome the problem of *process deceptive idleness* [BBG+99, ID01] and facilitates requests that are physically close on the magnetic medium by avoiding superfluous arm movements. The risk entailed in I/O anticipation is obvious: if the process expected to post a follow-up request does not do so within the anticipation interval, then the scheduler keeps the block device idle with no gain in return; so the time spent in anticipation essentially constitutes *penalty* that is clearly undesirable. However, successful anticipations are expected to frequently occur in many environments due to the following recurring I/O patterns:

- File-systems attempt to place logically adjacent blocks in physically successive sectors [MMF84, RO91].
- Many applications complete their entire block reading during the beginning of their process-lifetimes and flush their buffers near their termination phase creating distinct I/O activity periods [CDB04].
- Application servers, such as information and retrieval systems, typically produce multiple nearly-synchronous I/Os while evaluating queries [FNS95, OBRS94].

In this paper, we build on the success of *Anticipation Scheduler* [Lov05, ID01, Tra08], and propose an approach that uses anticipation intervals of variable length. The goal is to wait shorter in unsuccessful anticipations and minimize the penalties involved.

Our approach monitors all I/O requests of each process and by utilizing its most recent ones, dynamically places the process in question to a specific class. This class represents the process' recent behavior as implied by its block-device I/O access pattern. By designating an appropriate anticipation interval to each class, we can treat similarly behaving processes in a near-optimal way. In our proposal, we keep track of I/O request inter-arrival times as well as the distances between the sectors requested. Such spatio-temporal behavior of each process is taken into account in order to accurately establish its history and capture its specific nature (class) at a given time. In effect, as a process develops, its history may change. Our approach responds to this change by re-establishing the class in which the process may currently belong to. We minimize *AS*-related penalties by having the same process being treated with potentially different anticipation periods during its lifetime.

The proposed mechanism introduces a light-weight in-kernel element that cooperates with the existing anticipation policy and hints on the appropriate anticipation interval. Moreover, the in-kernel element is supported by two infrequently-invoked user-space components:

1. The *Classification Subsystem* that produces a classification scheme of all active processes.
2. The *Designation Subsystem* that assigns optimal anticipation intervals to each class of processes. This involves expressing the problem of assignment as a linear-optimization one and solving it.

As processes morph ever-changing workloads over time, the two user-space components help dynamically designate new classes of processes and provide appropriate anticipation periods. This outcome is fed to the kernel on-the-fly and so the disk scheduler may adapt to versatile circumstances the block-device might face. In this context, we term our approach as *Workload-dependent Anticipation Scheduling (WAS)* [1]. The precondition for the realization of anticipation periods with varying length is that the clock of the underlying architecture provides sufficient granularity. This is the case with all dominant contemporary architectures such as i386, x86-64 and sparc64 as well as many older ones including alpha and ppc [PH07, Car07].

We have developed the pertinent *WAS* kernel elements and associated user-space modules and have ported them into *Linux* running kernel *v.2.6.23*. Moreover, we have implemented an in-kernel *AS* scheduling policy that realizes the 95% rule for the setting the length of the anticipation intervals [ID01]. Using the above two implementations along with the standard *Linux AS* (*LAS*) implementation that features a fix-anticipation period, we have experimented with a wide range of workloads. Our results establish that *WAS* offers anticipation with shorter penalties and fewer waits than its counterparts. In addition, *WAS* better identifies the nature of the processes that make up a workload, is more adaptive to changing process behavior, works seamlessly with kernel mechanisms such prefetching and overcomes software/hardware changes in the underlying computing system more gracefully. The rest of the paper is organized as follows: Section 2 outlines the existing *Linux* kernel for disk scheduling. Section 3 presents our approach in providing anticipatory scheduling with varying intervals that are workload-dependent. Section 4 discusses our experimental evaluation and findings while Section 5 briefly presents related work. Conclusions can be found in Section 6.

2 Disk-Scheduling in the *Linux* Kernel

The current *Linux v.2.6.23* kernel offers an array of disk scheduling options that include the *Deadline-I/O, Anticipatory, Complete Fair Queuing* and the *Noop-I/O* schedulers. In prior releases, the *Elevator* discipline was the default disk-scheduling policy and featured front and back-merging of incoming requests to the queue of pending disk jobs according to their sector-number. However, highly localized disk access patterns rendered this pure *Elevator* discipline rather ineffective creating starvation at times.

The *Deadline-I/O* discipline favors *readers* over *writers* [Lov05, BC05]. It employs a differentiating treatment for these two types of block requests as *reads* are assigned a default expiration time of 500 milliseconds and *writes* a 5 seconds expiration time. The notion of deadline here is different from that in more traditional real-time settings [BB94, BSR88, HLC90] as it offers no guarantees about strict job latency. The *Deadline-I/O* approach uses three queues: the Read and the Write Queues accept in FIFO respective requests. All requests also enter the Elevator Queue as Figure 1 depicts and are sorted according to their sector-number. In the usual mode of operation, the *Deadline-I/O* scheduler services the first request from the Elevator Queue. Once, this request is dispatched to the device, its corresponding entry in the FIFO queues is also removed. Should a request at the head of either Read or Write Queue have waited longer than its designated threshold (i.e., more than 500 milliseconds or 5 seconds respectively) and is currently *expired*, it automatically has the highest priority and gets immediately dispatched for service. In addition, its corresponding entry from the

[1] pronounced "vas".

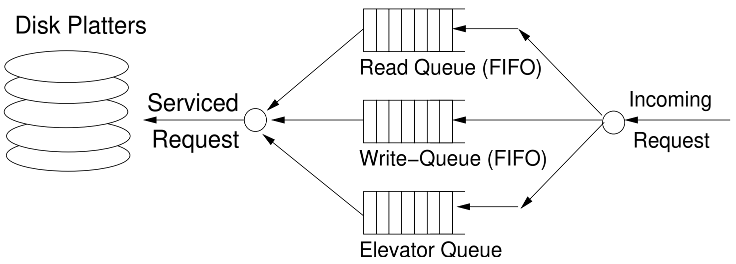

Figure 1: *Deadline-I/O* Scheduling

Elevator Queue is removed. The assignment of longer expiration intervals to *write* requests than their *read* counterparts ensures that the latter enjoy the best chances for quick service. *Writers* do not starve as they are always served once their expiration occurs.

The *AS* scheduling that is currently part of the *Linux* kernel *v.2.6.23* extends the operation of the *Deadline-I/O* scheduler by potentially anticipating the *read* requests. In its fundamental underpinnings, the *Linux Anticipation Scheduling (LAS)* uses all elements of Figure 1. Each request obtains an expiration time within which it should be dispatched to the block-device. Should any request located in either Read or Write Queue expire, the scheduler "rushes" to dispatch it. Through this aging feature, *LAS* properly handles the starvation of *writes* apart from avoiding unnecessary seeks when multiple *read* requests for nearby blocks are posted. As far as anticipation is concerned *writes* do not receive any special care but *reads* do. Any time a *read* request is serviced, the kernel determines on the fly whether to wait for a short period of time in anticipation of another nearby I/O launched by the same process. If the anticipation does eventuate and the I/O finally takes place, the kernel dispatches the newly-posted I/O instead of handling the item at the head of the Elevator Queue. In doing so, *LAS* can help further improve the block-device throughput for a wide range of file system workloads [Lov05, ID01].

It is worth discussing the subtle mechanism that determines whether the disk has to stall. Each process maintains two pieces of estimated information which are: a) its mean think-time [RF77], and b) its mean seek-distance traveled between consecutive requests [LZGS84]. The kernel enters anticipation only when the following two conditions are met:

1. The current mean think-time of the just serviced process is less than the set anticipation interval.
2. The mean seek-distance of the just serviced process is less than the distance to the request at the head of the *Elevator Queue*.

In *LAS*, the anticipation interval during which a device stalls is fixed and remains unchanged throughout the operation of the kernel. Its duration is a product of empirical observation [Lov05, BC05] and is constrained by the kernel clock which in turn depends on the hardware that the OS runs on. Since *Linux* supports a great range of architectures the frequency of the kernel clock is a kernel build time parameter. In a wide range of architectures including i386, alpha, ppc, sparc64, m88knommu and x86-64, the kernel clock can be tuned to tick once every millisecond. In this case, the anticipation period is set to a fixed $6\,ms$ period (or 6 clock ticks). Our proposed approach does require such clock granularity in order to properly set the duration of the anticipation intervals. Another group of hardware architectures consisting of arm, sparc, mips and m68k have their clocks tick only 100 times per second and so the anticipation period is set to at least $10\,ms$ (or 1 clock tick). The selection of the *LAS* anticipation period plays a major role in minimizing seeks and maximizing block-device efficiency.

Apart from *Deadline* and *Anticipatory* schedulers, the *2.6.23 Linux* kernel offers two more I/O scheduling alternatives: *Complete Fair Queuing I/O (CFQ)* and *Noop I/O*. The *CFQ* scheduler attempts to equally share the disk I/O bandwidth among all processes placing requests, therefore it is suitable for desktop and multimedia applications. For each process submitting requests to a block-device there is a distinct queue. In this manner, requests from the same process are placed in the same queue sorted according to their sector number. *CFQ* services a number of requests from each queue (by default four) before proceeding in serving the next queue in a round robin fashion. To this effect, *CFQ* safeguards per-process fairness. The *Noop I/O* scheduler simply coalesces a request with an adjacent one if there is such an opportunity. Beyond this, the *Noop I/O* does nothing more than serving in strictly *FIFO* fashion and is intended for use with flash memory devices where accessing any block presents the same time overhead.

Finally, the disk scheduling of *Linux* is supplemented with a prefetch mechanism. Prefetching synergistically works with disk scheduling and is a must-have feature for contemporary OSs [SGG03, J. 00, MNN03]. In *Linux*, prefetching mostly deals with sequential accessing of blocks and its function is based on a window of pages to be read "ahead" of time. The size of this window dynamically adjusts to specific application needs [Lov05]. Both *LAS* scheduler and prefetching are fine-tuned elements of the kernel and operate in tandem. While we provide an enhanced approach for anticipatory-based scheduling, our intention is not to examine the benefits of our proposal in isolation but rather establish its strength in operational systems. To this end and during our evaluation, we do not carry out simulations. Instead, we perform all our experiments involving pragmatic workloads on actual systems with the vital mechanism of prefetching either enabled or disabled. Our comparison involves our own proposal versus all available disk-scheduling options in the current *Linux* kernel.

3 The *WAS* Adaptive Anticipation Approach

In this section, we outline our approach that uses anticipation intervals whose length may vary over time. Through process monitoring, we establish a scheme that allows for workload-based selection of the anticipation period of each and every I/O anticipation in the kernel.

When in regular operation, *WAS* simply has to set the length of period during which disk stalls in hope of a successive request placed by the process just served. To determine the length of anticipation, *WAS* identifies the process behavior by examining its recent I/O history and associates the process with a specific class of behavior in a CPU light-weight fashion. The set of the classes in question as well as the anticipation intervals assigned to each class are products of a number of initialization steps taking place only infrequently. In the following, we present an overview of these steps as well as the operational features of the *WAS*.

3.1 Initialization Aspects of *WAS*

Before *WAS* commences its regular operation, a number of initialization steps must occur. Figure 2 depicts the flow of the operations and marks the input as well as the output of all phases involved. At first, *WAS* requires a *calibration phase* whose main objective is to produce a viable classification scheme. This scheme defines a finite number of N behavioral classes on which processes are eventually mapped to. The *calibration* phase is only required when there are major hardware or software changes; such changes include CPU, board, memory, disk upgrades as well as migrations to different file systems. Although we impose no minimum duration for the calibration process, we expect it to last for at least a few minutes [2]. During calibration, *WAS traces* all I/O requests that reach the kernel. At this time, *WAS* essentially functions as the standard *LAS*-scheduler but with anticipation period raised from $6ms$ to a maximum value (often $10ms$). The spatio-temporal differences of the I/Os in the trace along with minimal administrator input help jointly

[2] This has been actually our experience during our experimentation with *WAS*.

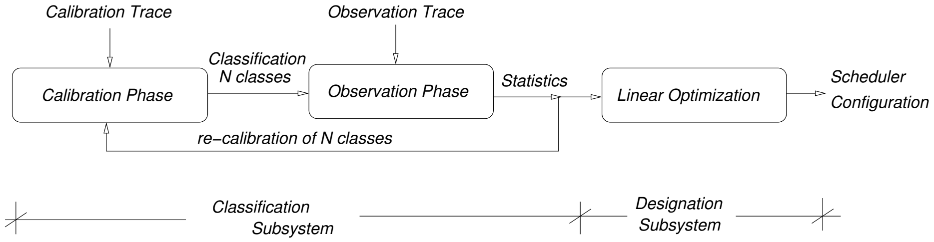

Figure 2: Flow of operations to configure *WAS* (resulting to scheduler configurations used during regular operation).

produce a set of distinct classes corresponding to different types of process behavior. Such a classification scheme is successful, should it place processes with different behavior to different classes. We point out that during regular *WAS* operation –discussed latter– the association of a process with a class is not fixed as a process may change classes throughout its lifetime.

Subsequently, *WAS* enters the *observation period* in which it still operates with a fixed anticipation interval (i.e., as *LAS*-scheduler). During this phase, we compile statistics regarding the successful/unsuccessful anticipations taking place within each and every of the adopted classes. These statistics guide the administrator to determine whether the adopted classification scheme is acceptable. If the classification scheme turns out not to be a plausible choice –mostly due to its inability to effectively differentiate among the process behavior classes adopted– the administrator can modify both composition and nature of classes and re-enter a new *calibration phase* (Figure 2). This procedure of setting up different process classes and then observing their effectiveness is iterative in nature and we use it to ascertain the plausibility of the finally adopted classification scheme and pertinent statistics.

We may directly use the aforementioned statistics to designate anticipation periods to process classes either manually or heuristically. However, such a course of action inevitably leads to poor anticipation interval assignment. Instead, we may deploy an *optimization phase* which addresses the problem at hand in a rigorous manner: using both the adopted process classes and their pertinent statistics, we solve a linear optimization problem and associate an optimal anticipation period for each process class (Figure 2).

In *WAS*, we realize the iteration over *calibration* and the *observation* phases via the *Classification Subsystem*. The latter is responsible for assembling the I/O trace, creating initial classification scheme, assessing the utility of the adopted classification ultimately re-adjusting and accepting of the classification scheme. Similarly, *WAS* carries out the *optimization phase* with the help of its *Designation Subsystem*. Both *Classification* and *Designation Subsystems* may be computationally-intensive and, thus, we deploy them as modules in user-space (Figure 3) invoked only infrequently [3].

The combined effect of the two subsystems, as illustrated in Figure 2, is a scheduler configuration, that represents the full adaptation of the *WAS*-scheduler to the characteristics of the hardware and the encountered workload. A scheduler configuration consists of the selected classification scheme (fed into the *Classifier Element* of Figure 3) and the designated anticipation periods for all process classes (that populate the *Anticipation Lookup Table* of Figure 3). If workloads are recurrent over specific periods of time, such scheduler configurations can be (re-)used in a "plug-and-play" fashion by the administrator. Similarly, if a group of identical machines —say in a cluster— handle the same type of workloads, a scheduler configuration can be cloned across all computing systems.

[3] In Section 4.6, we discuss the incurred overheads for the execution of these components.

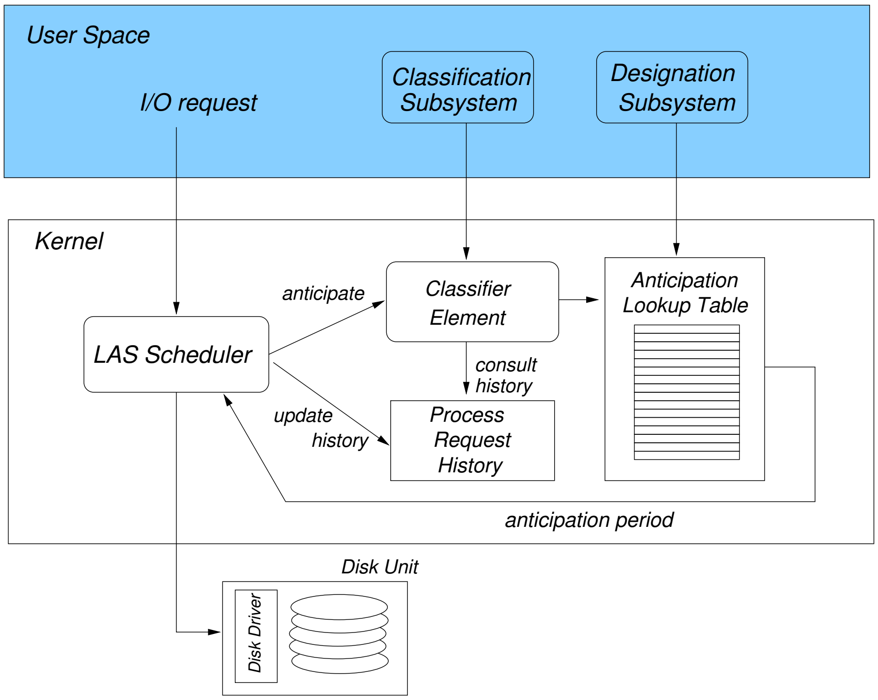

Figure 3: The *WAS* approach: refining the operation of the *LAS*-scheduler.

3.2 *WAS* Foundation Elements

The current *Linux* kernel "ignores" the block request history of a process as far as the length of the anticipation interval is concerned. It is upon this past history that we build our approach, and our conjecture is that minimal use of spatial and temporal elements of I/O request history can improve the effectiveness of *LAS*. Figure 4 depicts a typical time-line of a process consisting of CPU-bursts interleaved with I/O operations and periods of idleness. The latter is not shown for simplicity. For convenience, we designate the most-recent block request as I/O_i, the second most-recent as I/O_{i-1} and so on. Of key-importance here are the spatio-temporal differences between any two successive I/O-requests, namely:

- δt_i : The time difference between the *i*-th and the preceding (*i-1*)-th request in the past.
- δs_i : The space difference, in terms of sector distance, between the sector of *i*-th request and the sector of the preceding (*i-1*)-th request.

Evidently, the entire history of a process could be articulated by a series of such spatio-temporal differences of block requests.

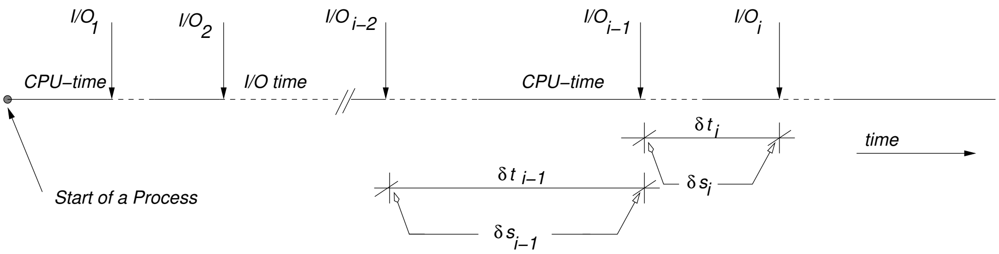

Figure 4: Layout of process request history.

We represent the spatio-temporal nature of a process history with a cyclic list whose elements entail the two types of differences. This list is linked to the in-kernel process structure so when the process terminates the memory resources used to store the list are freed. We use k-1 nodes in order to articulate the pair-wise relationships of the k most-recent block requests as Figure 5 shows. Initially, all nodes contain zero values. To compute a new pair of differences, we need to maintain per process the *last time-stamp* at which a request was posted and the *sector* of the request in question. Once all k-1 list elements are populated, the "oldest" one is replaced by the newest entry. A pointer helps mark the most recently entered pair of spatio-temporal differences. In the course of updating the list, the pointer always moves to the *top* of the structure. The kernel overhead needed for updating the process history is minimal as only a few instructions have to be executed any time a process issues a new I/O. Provided that history lists are maintained in kernel space, their length has to be bound. As the more-distant past history reveals less information about a process current behavior, the older portion of history becomes less critical when it comes to characterizing the inter-arrival and location aspects of ongoing I/Os [AS95, Den68]. Evidently, the history length presents a trade-off between memory-efficiency and accuracy of process characterization in terms of I/O activity.

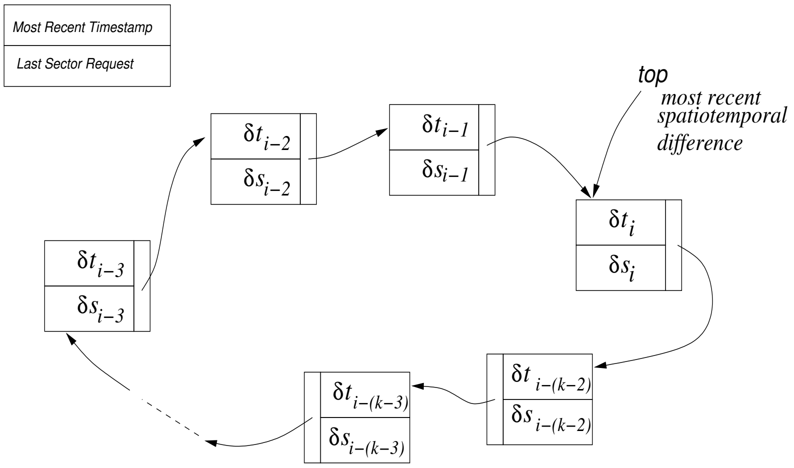

Figure 5: Structures pertinent to spatio-temporal history of a process.

WAS consults with the history of a specific process and suggests a proper *anticipation period*, any time the anticipatory-scheduler decides to stall for a forthcoming I/O. The duration of the *anticipation period* may vary over time for the same process; this is in contrast to *LAS* that stalls invariantly for a fixed period. *WAS* designates the appropriate anticipation interval in two steps:

1. The process whose I/O was just completed is mapped to a specific behavior class based on its recent I/O activity. This is determined on-the-fly by the *Classifier Element* (Figure 3) with the help of *Process I/O History*, the structure of which appears in Figure 5.
2. *WAS* uses the class number the process gets mapped to as an index to the *Anticipation Look-up Table* of Figure 3 and retrieves the corresponding anticipation period.

The in-kernel elements of *WAS* rely on the *Classification Subsystem* to produce the set of *classes* for different types of process behavior. Each class is expected to have different potential for an upcoming I/O, therefore is best served by a distinct anticipation period length. Moreover, *WAS* exploits the optimal anticipation periods produced by the *Designation Subsystem* for each process class. This information is stored in the *Anticipation Look-up Table* maintained in kernel space as Figure 3 shows. Note, that a process may be reassigned as far as its class is concerned due its own changing I/O request pattern, enabling the *WAS* scheduler to treat the same process differently in the course of time.

The assignment of a process to a class and the table look up, involved in anticipation interval selection, present minimal CPU overheads and, thus, both tasks are included in the kernel. On the other hand, we realize *Classification* and *Designation Subsystems* as user-space processes as both are computationally intensive and produce sizable main-memory footprints [Den68, PS82, TK05]. In what follows, we outline the work of the in-kernel *Classifier Element* in Section 3.3 and discuss in detail the operation of *Classification* and *Designation Subsystems* in Sections 3.4—3.6.

3.3 The *Classifier Element*

The primary goal of our in-kernel *Classifier Element* is to determine the "class" that a process currently belongs to. This is accomplished with the help of the process history and the notion of the *Anticipation Prospect Functions (APFs)*. *Anticipation prospect* estimates the potential for a successful anticipation based on a single spatio/temporal difference in the process history. With values in the [0..100] range, the *APF* represents the likelihood that a submitting process will benefit from anticipating a forthcoming block request. More specifically, an *anticipation prospect* value close to 100 promotes anticipation in the disk scheduler, as the forthcoming request is expected to be nearby. On the other hand, *anticipation prospect* values close to zero indicate that an upcoming I/O is rather unlikely to obtain gains from anticipation.

Our approach features two anticipation prospect functions : the *spatial-APF* based on sector distances in the I/O request history and the *temporal-APF* based on time differences. Figures 6 and 7 depict examples of such functions. Figure 6 shows the spatial anticipation prospect function over the difference in disk address space between subsequent I/O requests. Here, spatial anticipation prospect is a step function that discerns between three cases in spatial differences: small sector differences (0-40) that are promising for anticipation, medium sector differences (41-60) that are less promising, and large sector differences (61 and above) that are poor candidates for anticipation. In the same spirit, Figure 7 presents a sample temporal anticipation prospect function for time difference between requests in milliseconds. In general, large (spatial or temporal) differences indicate process behavior that is less promising for successful anticipations, therefore monotonically decreasing step functions are good candidates to be used as *APFs*.

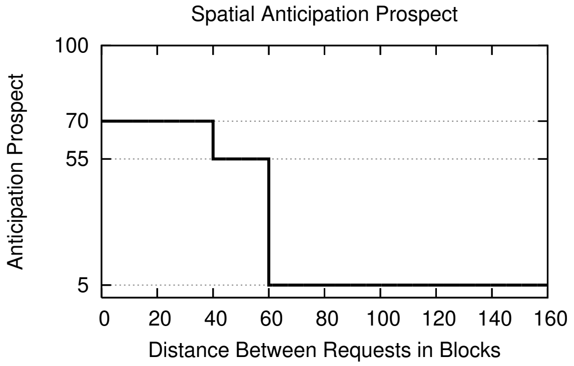

Figure 6: Sample of spatial anticipation prospect function $S(\delta s)$.

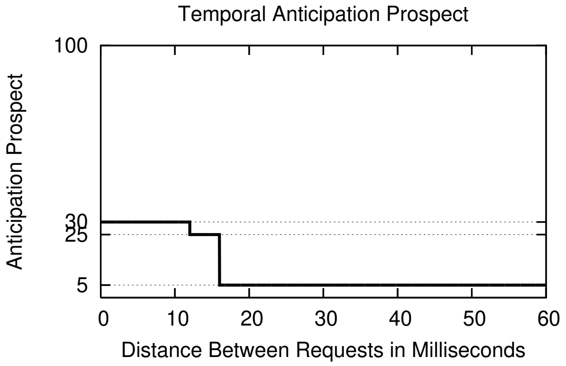

Figure 7: Sample of temporal anticipation prospect function $T(\delta t)$.

In our approach, *APF* functions are a product of the *calibration* phase and they are part of our scheduler configuration. Once available, the functions are jointly used by our in-kernel *Classifier Element* to dynamically map processes to classes. In particular, the spatio-temporal differences of a history entry j in Figure 5 contribute to the *combined* anticipation prospect value C (of equation 2) by factor:

$$C_j = S(\delta s_j) + T(\delta t_j) \tag{1}$$

where S and T are the respective spatial and temporal *APFs*. Note that C_j theoretically takes values in $[0, 200]$. In practice though, functions S and T may be such that they cannot sum up to 200, therefore, C_j takes values in $[0, \max\{C_j\}] \subseteq [0, 200]$. For instance, $\max\{C_j\}$ can be at most 70+30=100 if functions S and T of Figures 6 and 7 are used. In essence, C_j represents the prospect of successful anticipation, as estimated by a single spatio-temporal difference in the process history. The *Classifier Element* considers recent requests to be more significant than dated ones, as they often offer a better insight regarding the "intention" of a process. More specifically, our *Classifier Element* employs a geometric weighting scheme used to represent decay [AS95, CH81, Den68, FNS95]. If w_j is the weight of the j-th history entry, then the weight for the previous entry is $w_{j-1} = w_j/2$. So, the *Classifier Element* sums all C_j and normalizes the result to have the process at hand binded to one of the N classes at any given time.

In short, the output of the *Classifier Element*, that is the class number the process currently belongs to, is computed as follows: first, the classifier considers the entire process history in the weighted sum $\sum_{j=i}^{j=i-(k-2)} w_{j-i} C_j$ where i is the identifier of the most recent I/O request and the process history involves *k-1* elements as Figure 5 indicates. Subsequently, the sum is normalized in the range of $[0, 1]$ by dividing with $\sum_{j=i}^{j=i-(k-2)} w_{j-i} \max\{C_j\}$ which is the maximum value the weighted sum C_j can assume. Provided that *WAS* works with N classes of processes, expression (2) denotes the identifier of the class that the requesting process belongs to at this time:

$$C = \left\lceil N \cdot \frac{\sum_{j=i}^{j=i-(k-2)} w_{j-i} C_j}{\sum_{j=i}^{j=i-(k-2)} w_{j-i} \max\{C_j\}} \right\rceil \tag{2}$$

Evidently, C takes values in the range $[1, N]$.

The intuitive choice for N would have been a large number as more classes would help us specify process behavior more accurately. However, the number of classes is constrained by two factors: **a)** the size of the *Anticipation Lookup Table* that is a kernel structure and has to be as compact as possible and **b)** the computational needs of the *Designation Subsystem.* During our assignment of anticipation periods to classes that we discuss in Section 3.6, we solve a linear optimization problem. The complexity of this problem is, among other things, a function of the number of classes used. In practice, we stipulate that a few hundreds of classes (i.e., 100-300) is a good compromise between process behavior granularity and CPU costs involved.

3.4 The *Classification Subsystem*

The prime objective of the *Classification Subsystem* is to produce the two *APFs* ultimately used by the *Classifier Element* to map an I/O requesting process to a specific class (Figure 3). The secondary objective is to compile pertinent anticipation statistics for all suggested classes during the *observation phase* (Figure 2). The *APF* functions jointly form a classification scheme that may clearly differentiate among types of processes that either favor or do not favor consecutive I/O requests. Creating such a scheme is a known challenge [MNN03, TK05, KM05] and so we resort to both heuristics and experimentation to offer a solution that is inexpensive to generate, yet works well in practice.

To derive an *APF*, we commence from a template anticipation prospect function –such as the one shown in Figure 8– that eventually evolves into a specific function. Such an *APF* function template entails a number of steps m with decreasing values along the y-axis as the spatial or temporal delta increases along the x-axis. To produce a concrete *APF* function, we have to deal with three key template parameters: the number m

of steps involved, the y-value of each step, and the width of each step along the x-axis. Our approach takes as input the number of steps m and for each such step its respective y-value [4]. In this manner, the number of possible alternative prospect functions is limited.

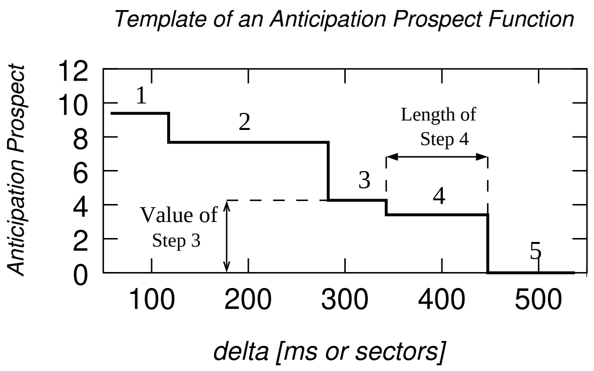

Figure 8: Template of an *Anticipation Prospect Function (APF).*

To determine the length of each step along the x-axis, the *Classification Subsystem* uses the I/O trace captured during the *calibration* phase. Figure 9 depicts the structure of trace data as a tuple. Requests are logged according to their arrival time and include the identifier of the requesting process, whether the I/O was anticipated, the first sector as well as the number of all subsequent sectors read, and finally, the service time that the kernel took to complete the I/O. With the above trace at hand, we determine the

Arrival_Time	*Service_Time*	*1st_Sector_to_Read*	*Num_of_Sectors_to_Read*	*Anticipated(Y/N)*	*ProcessID*

Figure 9: Data collected for every I/O (read) during the *trace* period of the *calibration* phase.

length of each step of the *APF* functions. Consider the case of the *spatial–AFP* for the sake of discussion. Each logged I/O maintains a distance in terms of sectors from its previous counterpart in the same process. Should we place all such spatial I/O differences from all processes in the trace on a graph, we create a simple distribution map whose x-axis corresponds to deltas in terms of number of sectors and the y-axis shows the observed population of each delta for the entire trace.

Algorithm 1 computes the *spatial-APF* using the above simple distribution map as input [5]. This is accomplished by computing the length of all the *APF* steps so that the populations appearing in all steps are nearly-equal (i.e., yielding a *near-equal-populated histogram* [Man84]). The length of each such resulting step quantifies the respective delta in the *spatial-APF.* The algorithm initially computes the number of spatial differences that every step would have, should all steps be equi-populated. This "exact" population -stored in *population_target*– is used as a target by the algorithm which gradually increases the step-length so that the resulting step-population –stored in *real_step_population*– reaches at least this target. The inner **while-loop** performs the increase of the step length while the outer **for-loop** iterates through all steps. As a prior step may be assigned more differences than its exact target population, the algorithm recomputes the *population_target* variable each time the length of a new step is to be assigned. Such minor deviations from equi-populated steps are acceptable as individual *APFs* do not have to be fine-grained. When we aggregate two anticipation prospect functions, we inadvertently create a large number of distinguishable values that the factor C_j of equation (1) may take. It is worth noting that the final class designation as depicted by

[4] These values are all user-defined.
[5] it can also compute the *temporal-APF* given the respective distribution map of temporal differences.

Algorithm 1 Determining the Length of each step in the *APFs*

Input:
`m`: Number of steps to be produced,
`NumOfDiffs`: Number of differences,
`DistMap [ ]` : Distribution map of all differences. Each item of the array is the population of the corresponding difference delta.

Output:
`StepLength [ ]` : Array of step lengths

Begin

$d = 1$ {pointer iterating through `DistMap`}
$cur\_step\_start = 0$ {where does the current step starts}
$remain\_diffs =$ `NumOfDiffs`
for $step = 1$ to `m` **do**
  $population\_target = remain\_diffs/$ (`m`$- step + 1$)
  {desired population of current $step$, it will be exceeded so we re-calculate for each step}
  $real\_step\_population = 0$
  while $population\_target > real\_step\_population$ **and** $remain\_diffs > real\_step\_population$ **do**
    $real_step_population += $ `DistMap[`d`]`
    $d += 1$
  end while
  `StepLength[`$step$`]` $= d - cur\_step\_start$
  {make preparations for the next step}
  $cur\_step\_start = d$
  {calculate remaining differences to equally distribute among the rest of the steps}
  $remain\_diffs -= real\_step\_population$
end for

End

equation (2) is a function of the two prospect functions and the aging of I/O requests.

3.5 Ascertaining the Choice of the *APFs*

While operating in *observation* period, *WAS* can help the administrator ascertain the quality of the choices made regarding *APFs*. To this end, *WAS* monitors how each class behaves when a maximum fixed anticipation period is used and accumulates statistics for every class of the scheme defined via the adopted *APFs*. The statistics in question pertain to the successful and/or unsuccessful anticipations per class.

During *observation*, for every anticipation attempted within every class we monitor the time elapsed between the initiation of an anticipation and its ultimate success or failure. Therefore, at the end of the *observation* period we know the specific statistics of I/O requests that either have succeeded in their anticipation or failed. Figure 10 presents the number of failed and succeeded anticipations per class when the *Cold Boot* workload discussed in Section 4 is used. Here, the *APFs* are derived using the *Cold Boot* workload in the *calibration* phase and we also assume N=100 classes. In a more detailed view, Figure 11 depicts the compiled statistics for the specific class "*28*" of Figure 10. In this, 120 of I/O requests eventuate only after stalling for just $1ms$, around 100 requests complete after stalling for $2ms$ etc.; the depicted right-most column shows the amount of anticipations that ultimately failed. We name the above accrued per-class statistics as *Anticipation Distribution (AD)*. We note that all *N=100* classes present corresponding *ADs* as that of Figure 11.

Anticipation Distributions (ADs) serve two purposes:

1. They help the administrator ascertain the utility and the success of the selected-thus-far classes of processes. If the *ADs* at hand indicate a poor selection of *APFs*/classification-scheme, the user may have to step back, intervene, and start another *calibration* phase so that new and presumably better *APFs* are selected.

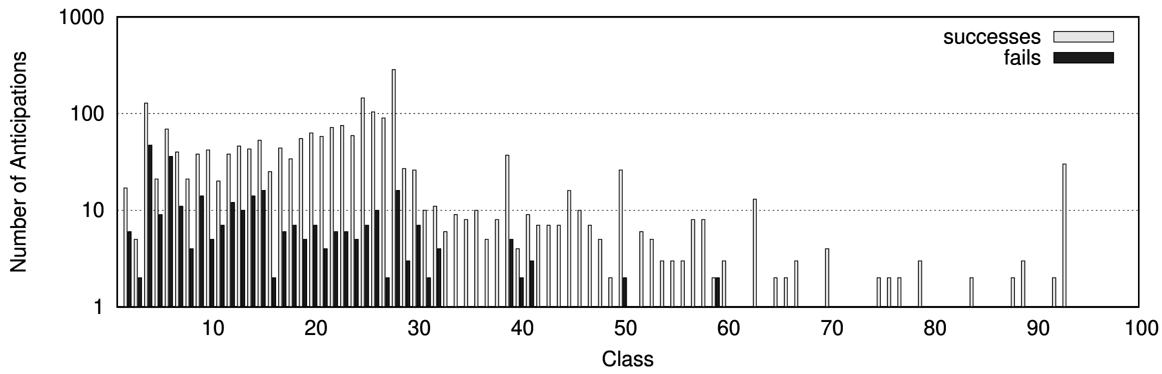

Figure 10: Number of successful/failed anticipations for N=100 classes–*Cold Boot* workload.

2. They are fed into the *Designation Subsystem*, should they portray a good distribution of process classes. The *Designation Subsystem* optimally assigns a specific anticipation period to every class. With this assignment completed, the *Anticipation Lookup Table* of *WAS* is populated and the scheduler commences its normal operation.

In what follows, we elaborate the first of the above points while we discuss the second in Section 3.6.

ADs help differentiate viable choices made regarding the classification scheme from poor ones. Apparently, the worst feasible classification is the one that consists of a single class as it cannot discriminate among different process behaviors and all classes are treated in the same manner. Figure 12 shows the *AD* of such a single-class scheme used to characterize the behavior of I/Os in the *Cold Boot* workload. Interestingly, this is the case with *LAS* as all I/Os are indiscriminately stalled for $6\,ms$. An equally unhelpful scheme is the one where multiple classes exist but all articulate the same behavior, thus, providing no insight for better anticipation. In this case, there is no reason to treat any class differently as the corresponding *ADs* offer no differentiation information. Consequently, the more evident the distinction among classes is, the better the classification scheme ultimately is. For instance, in Figure 10 we see that some classes, such as class "*28*", feature far more successes than fails; conversely, other classes, such as class "*2*", appear to have only a few more successes than fails. Such clear distinction between classes and respective *ADs* indicates successful *calibration*. Finally, another unsuccessful classification results when many classes feature no anticipations. Depending on the nature of workloads used during *calibration*, some classes are by far more populous than others due to recurring I/O access patterns. However, classifications that populate only few classes are undesirable since scarcely populated classes represent rare process behaviors. In short, *ADs* that may prompt *re-calibration* and call for the re-adjustment of *APFs* demonstrate one or more of the following properties:

- all *ADs* are similar to the distribution derived by the single-class classification scheme,
- no differentiation exist among the *ADs* of different classes,
- there are many scarcely-populated classes.

The administrator's intervention would likely entail the use of a different *trace* and/or changes in the number of steps m while setting the *APF*-functions and their y-axis corresponding values. In general, a *trace* should include processes that demonstrate as much diverse behavior as possible so that a more successful classification scheme is established. As *re-calibration* is somewhat of an involved procedure, it is meant and actually takes place infrequently.

It is worth pointing out that the *re-calibration* could be further automated so that administrators do not have to delve into details. To this end, we could use a similarity metric such as the correlation coeffi-

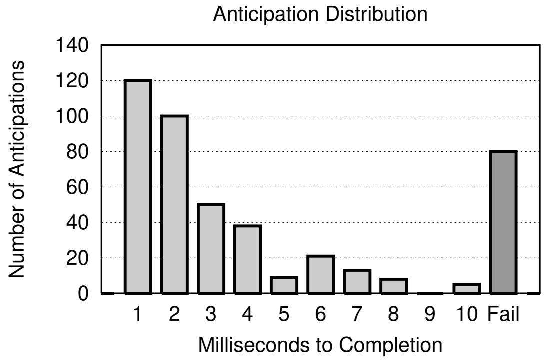

Figure 11: *Anticipation Distribution (AD)* for the specific class 28 from the *N=100* classes involved in the example *Cold Boot* workload classification.

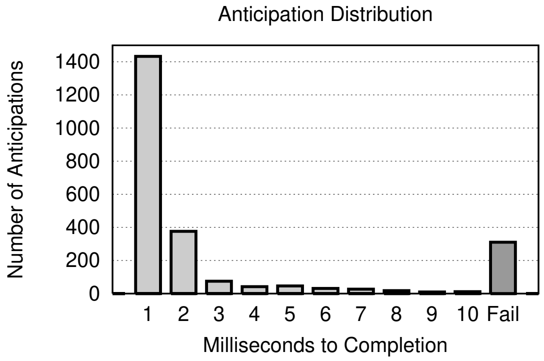

Figure 12: The *Anticipation Distribution (AD)* derived after observing the entire *Cold Boot* workload, assuming a single process class and fixed anticipation interval at $10\,ms$.

cient [Man84] to measure the "similarity" among all classes adopted in a workload and use this as a criterion to re-adjust the *APF*s. In particular, we could represent each *AD* as a vector. For example, the *AD* representing class 28 depicted in Figure 11 would be *<120,100,50,38,9,21,13,8,0,5,80>*. Then, the coefficient similarity between the *AD* of each class and the *AD* for the entire workload (i.e., single-class classification scheme such as that depicted in Figure 12) can be computed. Having derived N such values, we could then compute the average similarity value for the classification at hand. This value should be definitely lower than one and should ideally be as close as possible to zero. Average value one indicates classes with identical behavior and zero exemplifies a set of completely different classes something which is very difficult to attain in pragmatic settings. Moreover, we could assist the automated process in discussion by requiring that no more than μ% of classes are empty. Although a *re-calibration* process following the above approach can be developed, its realization falls beyond the scope of our work here and we plan to examine it in the future. We have followed a semi-manual approach instead and while working with the *Cold Boot* workload and imposing that no more than 30% of classes are without any population, we established an average similarity value of 0.63. The latter represents a viable and pragmatic choice for the *AD*s finally adopted in one of the classifications used.

3.6 The *Designation Subsystem*

The objective of the user-space *Designation Subsystem* is to determine the appropriate anticipation interval for every class produced by the *Classification Subsystem*. We treat this as an optimization problem which involves the N process behavior classes and the respective *anticipation distributions (ADs)* produced by the *Classification Subsystem*.

Before laying out this optimization problem, we consider the case of the class characterized by the distribution of Figure 11. In an attempt to designate the best possible anticipation interval, we may start reducing the stalling duration; should we change it to $6\,ms$ the failed anticipations will increase to include those currently in the 7–$10\,ms$ range. At the same time, the shorter anticipation period will reduce the *penalty time* for keeping the disk idle. We define *penalty time* as total waiting time for I/Os that took the chance to enter anticipation but did not eventuate. While assigning specific anticipation intervals to classes, we have to deal with two contradicting goals. Namely, we have to:

- lower the number of failures,

- minimize the overall penalty time.

Minimizing the total time the hard disk remains idle in anticipation of a request favors shorter anticipation periods. In contrast, lowering the number of failures calls for longer anticipation periods. In our example, we let the anticipation intervals range between 1–10 ms and show the above contradicting objectives with the help of Figures 13 and 14. The two graphs depict the decreasing number of failed anticipations and the

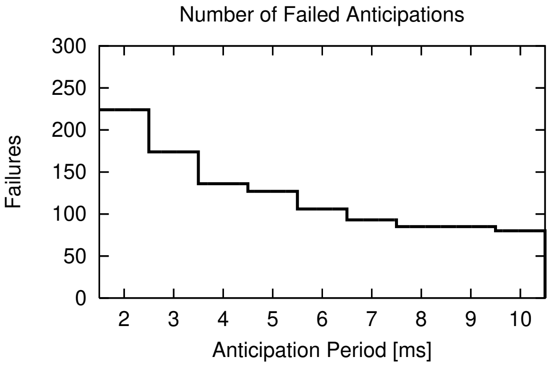

Figure 13: Failed anticipations for different anticipation periods.

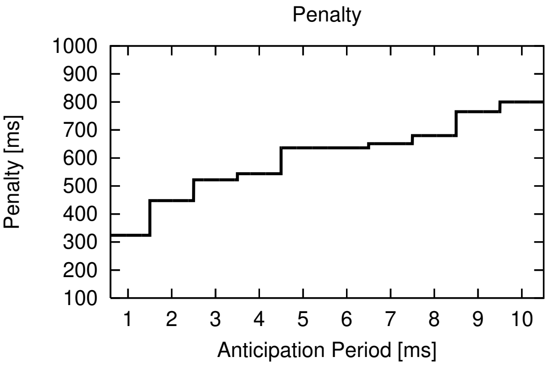

Figure 14: Anticipation penalty for different anticipation periods

increasing corresponding penalties as we augment the anticipation interval. In order to work a compromise between the two objectives, we could suggest the use of the *average penalty per successful anticipation* heuristic. Figure 15 shows how values for this heuristic range for values obtained from the distribution of Figure 11. Here, the "best" anticipation period for the class of Figure 11 appears to be 4 ms. However in the

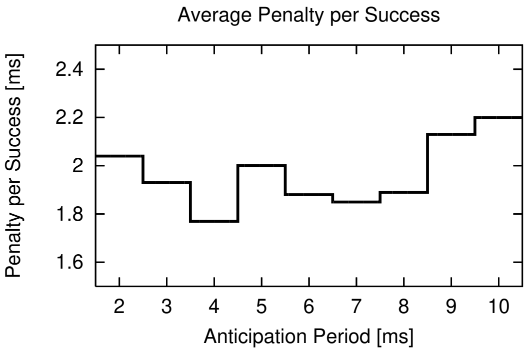

Figure 15: Average penalty per success for different anticipation periods

context of the *Designation Subsystem*, we address the anticipation interval assignment more rigorously by treating it as a linear optimization problem; this is done so that an overall optimal solution is found. This optimization problem is carried out in user-space only when the administrator wishes to fine-tune *WAS* to a specific workload. The objective function we seek to minimize is the total number of failed requests for all N classes:

$$\min \left\{ \sum_{i=1}^{N} Fails_i \right\} \tag{3}$$

Our linear optimization exclusively relies on the information that the *Anticipation Distributions (ADs)* offer. This information in *ADs* can be readily represented by a matrix $A[i,j]$ featuring N rows and $M+1$ columns. The rows correspond to the N process behavior classes adopted in the classification scheme and M

is the number of different milliseconds along the x-axis of the ADs. Each column identifier reflects the number of milliseconds elapsed in successful anticipations. The last column of the matrix provides the number of observed failed anticipations per class during the classification phase. In A, values $S_{i,j}$ indicate the number of successful anticipations observed for class i during the j^{th} ms of the anticipation period; values F_i denote the observed failures for class i when intervals are longer than $M\,ms$. Therefore, we define:

$$A[i,j] = \begin{bmatrix} S_{1,1} & S_{1,2} & \cdots & S_{1,M} & F_1 \\ S_{2,1} & S_{2,2} & \cdots & S_{2,M} & F_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ S_{N,1} & S_{N,2} & \cdots & S_{N,M} & F_N \end{bmatrix}, \text{ with } i \in [1,N] \text{ and } j \in [1, M+1]. \tag{4}$$

Assume that the *Designation Subsystem* assigns class i to an anticipation interval that lasts 5 ms. Then, according to matrix A the number of failed anticipations will be the sum of all elements in row i, from column $5+1$ to the last column $M+1$, that is $\sum_{k=6}^{M+1} A[i,k]$. This specific cumulative information for all classes is presented by matrix B as follows:

$$B[i,j] = \begin{bmatrix} \sum_{k=2}^{M+1} A[1,k] & \sum_{k=3}^{M+1} A[1,k] & \cdots & \sum_{k=M+1}^{M+1} A[1,k] \\ \sum_{k=2}^{M+1} A[2,k] & \sum_{k=3}^{M+1} A[2,k] & \cdots & \sum_{k=M+1}^{M+1} A[2,k] \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=2}^{M+1} A[N,k] & \sum_{k=3}^{M+1} A[N,k] & \cdots & \sum_{k=M+1}^{M+1} A[N,k] \end{bmatrix} \tag{5}$$

We denote the solution of the linear optimization problem with X, where:

$$X[i,j] = \begin{cases} 1 & \text{if class } i \text{ is assigned } j\,ms \text{ anticipation} \\ 0 & \text{if class } i \text{ is not assigned } j\,ms \text{ anticipation} \end{cases} \tag{6}$$

for $i \in [1,N]$ and $j \in [1,M]$. Evidently, we can only assign one anticipation period to a class, so:

$$\sum_{j=1}^{M} X[i,j] = 1, \qquad \forall \text{ class } i \in [1,N] \tag{7}$$

We revisit Expression 3 and rewrite our objective function in terms of X and B. Note that given the solution in X, the number of requests $Fails_i$ that are expected to fail for class $i \in [1,N]$ is $\sum_{j=1}^{M} X[i,j]B[i,j]$. Consequently, our objective expression becomes:

$$\min \left\{ \sum_{i=1}^{N} \sum_{j=1}^{M} X[i,j]B[i,j] \right\} \tag{8}$$

Should we try to solve the problem with the conditions provided thus far, we will obtain the longest possible anticipation time, that is M, for each class. This is expected since the maximum anticipation period will produce the fewest failed anticipations. As noted before, we have to consider two contradicting objectives. At this point, we introduce a constraint that expresses the quality of the solution which is directly related not only to the sum of all failed anticipations, but also to the time spent in anticipation, and, thus, incorporate the second objective in the linear optimization problem. We introduce this aspect by imposing the constraint that the average waiting time per failed request does not exceed a threshold:

$$\frac{Penalty}{Fails} < G \tag{9}$$

With this new parameterized rule, our solution can meet certain quality criteria: lower G yields solutions where anticipations occur only if there is high probability that they will succeed; similarly, higher G will return solutions where anticipations occur, even if their probability of success is not very high. However, we should bear in mind that enhanced quality, that is small values for G, makes the optimization problem harder and, thus, comes at the expense of CPU overhead during problem solving. Revisiting condition 9, we express $Fails$ as:

$$Fails = \sum_{i=1}^{N} \sum_{j=1}^{M} X[i,j]B[i,j] \tag{10}$$

For the $Penalty$, we need to consider that a class assigned anticipation period j ms also sustains a penalty of j ms for each failed anticipation. Consequently the penalty time for all classes is:

$$Penalty = \sum_{i=1}^{N} \sum_{j=1}^{M} X[i,j]B[i,j]j \tag{11}$$

By substituting Expressions 10 and 11 to Condition 9 the quality constraint is expressed as follows:

$$\sum_{i=1}^{N} \sum_{j=1}^{M} X[i,j]B[i,j](j - G) < 0 \tag{12}$$

Conditions 6, 7, 8, and 12 make up a linear system whose solution can optimally assign anticipation interval to classes. We efficiently solve this linear system using the Simplex method [PS82].

In some cases, the solution of this optimization problem suggests that some classes have anticipation period near or exactly zero. This is the case of classes where their respective *ADs* show mostly failed attempts. In general, having anticipation period close to zero cancels out the benefits of anticipation. Therefore, we may also impose that anticipation intervals feature values above a threshold q. Such a constraint would be:

$$X[i,j] = 0, \;\; \forall i,j : i \in [1,N], j \in [1,q] \tag{13}$$

Overall, the solution of this linear optimization problem allows for the flexible use of CPU cycles through constraint 9. At the same time, the assignment of the anticipation periods for all classes does not happen in isolation as is the case with the possible use of the *average penalty per successful anticipation* heuristic presented earlier in Section 3.6.

4 Experimental Evaluation

We have developed both the in-kernel and user-space modules of *WAS* in *C* and ported our implementation to *Linux* with kernel *v.2.6.23*. Moreover, we have implemented a *LAS*-scheduler that uses the 95%-rule for setting the length of the anticipation intervals [ID01]; although such a patch has been available for *FreeBSD* [Iye01], to the best of our knowledge our implementation is the first for *Linux*. The main objective of our experimental effort was to show that while *WAS* still enjoys the performance advantages of *AS*-based over traditional disciplines, it combats deceptive idleness with fewer operational penalties. During experimentation, we used a variety of both synthetic and application workloads and/or traces. These workloads represent a wide range of operational environments for block-devices and fall into three categories:

1. workloads produced by benchmarks,
2. workloads produced by processes that mimic specific types of system behavior, and
3. customized workloads that aim to reveal specific aspects of the operation of block-device schedulers.

The first group consists of the *Andrew's benchmark* [HKM+88], the *kernel compile* [Lov05] and the Bonnie++ workload [Cok08]. *Andrew's benchmark* was extended so that multiple instances of the benchmark are in concurrent execution. In this first group, the *kernel compile* and Bonnie++ are at the opposite ends of the spectrum: *kernel compile* involves accessing many small files and anticipation may hamper performance while Bonnie++ accesses different parts of large files. The second category entails workloads that attempt to recreate I/O traces encountered in multi-user systems, database servers and scientific applications [SS91]. The final set of workloads consists of the *Linux* OS "*Cold Boot*" and the synthetic *KVA*. Both serve as mechanisms to reveal strong and weak points in the *WAS* behavior. Our experimentation also investigates the impact prefetching and "slower" hardware may have on our approach. Before discussing key findings obtained for each workload, we present our methodology and the metrics used.

4.1 Methodology

As discussed in Section 3, the *calibration* and *observation* periods are critical in deriving viable classification schemes. Choosing an appropriate such scheme is crucial in capturing the main software and hardware features of the computing system at hand. Out of the eight workloads we present in this section, the *Cold Boot* and the *Andrew's-like benchmark* are versatile enough to yield viable classification schemes. These two workloads feature processes with diverse I/O behaviors and thus, they are good candidates for creating classification schemes. The *Cold Boot* reads numerous files requiring —in most of the times— one seek per file. Moreover, the 1 *GB* of main memory available in our system is sufficient to cache the entire boot procedure so blocks are read only once. The *Andrew's-like* benchmark is different as it seeks into many small files that exceed the capacity of the main memory causing the files to be buffered out. This in turn causes cache misses and forces block-device activity. Using these workloads for *calibration*, we generate two classification schemes for *WAS*. Each classification scheme yields a different population for the *Lookup Table*. A particular classification scheme along with its respective *Lookup Table* values constitutes a scheduler configuration.

For all eight workloads examined here, we experiment with four anticipatory disk-scheduling disciplines and/or scheduler configurations, namely:

1. *WAS(Andrew's-like)*: Our proposed *WAS* approach featuring varying anticipation periods and using a classification scheme derived via the *Andrew's-like* workload.
2. *WAS(Cold Boot)*: Similar to previous, but *WAS* scheduler now uses a classification scheme derived with the help of the *Cold Boot* workload.
3. *95% heuristic*: Our implementation of the heuristic described in [ID01]. This algorithm maintains statistics on the arrival time of requests per process. It then sets the anticipation time to be such that 95% of the requests, already served, would have been successful.
4. *Static* 6 *ms*: The default *Linux* disk-scheduler (*LAS*) that assumes a constant 6 *ms* anticipation interval.

We used two different computing systems in our experimentation: a *"fast"* one that was a *Pentium 4* at 3 *GHz* with 1 *GB* main-memory and 80 *GB* Serial-ATA hard disk and a *"slow"* one which was a *Pentium M* at 1.3 *GHz* equipped with 512 *MB* of memory and a 30 *GB* Ultra-ATA-100 hard disk. The *"fast"* system served as our main experimentation platform. For both systems, we used two classification schemes calibrated through the *Cold Boot* and the *Andrew's-like* workloads. Figures 16 and 17 show the respective *Anticipation Prospect Functions (APFs)* for the above classification schemes termed *Fast Cold Boot*, *Fast Andrew's*, *Slow Cold Boot* and *Slow Andrew's*.

The *spatial-APFs* are mainly affected by the pattern with which blocks in the specific workloads are accessed. As both slow and fast machines use *reiserfs* as their file-system, the same workload produces identical spatial access patterns in both systems. Consequently, the spatial anticipation functions are identical in both machines when the same workload (i.e., *Andrew's-like* or *Cold Boot* benchmark - Figure 16) is used. In

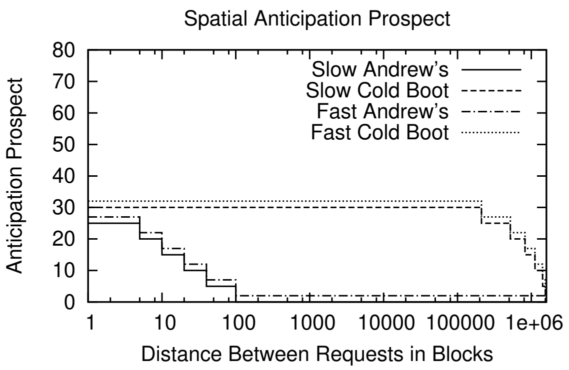

Figure 16: *Spatial-APFs* derived through *calibration* with *Cold Boot* and *Andrew's-like* workloads

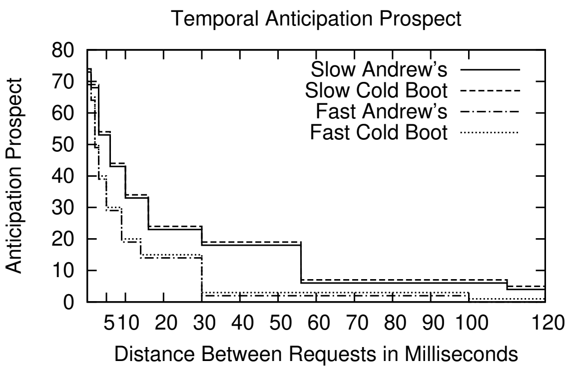

Figure 17: *Temporal-APFs* derived through *calibration* with *Cold Boot* and *Andrew's-like* workloads

similar spirit, the *temporal-APFs* are mainly affected by the frequency with which I/O requests are posted. This means that even different workloads would produce similar temporal associations on the same hardware (i.e., *Fast Andrew's* and *Fast Cold Boot* in Figure 17).

Through extensive experimentation, we established that a history length of five (5) I/Os sufficiently captures the behavior of a process for the examined workloads. In addition, we set the optimization goal G to 6 ms. As discussed in Section 3.6, G is the average waiting time that the *Designation Subsystem* optimization attempts to accomplish. By setting G to 6 ms, we impose that the average anticipation period in the *WAS(Cold Boot)* and *WAS(Andrew's-like)* configurations tries to be equal to the constant anticipation period of *Static* 6 ms allowing us the comparison with *LAS*.

4.2 Evaluation Metrics

Prior work has examined the effectiveness of *AS* versus that of conventional disk-scheduling policies by using throughput as the main comparison metric [ID01]. In our work, we have ascertained previous results on the performance of *AS*. We have also established that even if no varying anticipation periods are used –as is the case with *LAS*– *AS*-based schedulers consistently produce higher throughput rates than their conventional counterparts. Our main goal in this evaluation is to investigate the behavior of the various *AS*-based schedulers and in particular evaluate the effectiveness of those that use varying-length anticipation periods to combat deceptive idleness.

Extensive experimentation with anticipatory scheduling variations produced small gains in terms of throughput for *WAS* in comparison to *LAS* and *95%-heuristic*. In real-life OS deployment, there is a multitude of important factors that may interfere as far as throughput is concerned apart from the anticipation periods used. Most notably, throughput heavily depends on prefetch, cache mechanisms (implemented in both hardware and software) and of course on the mode the scheduler operates in. For instance as Section 2 points out, *LAS* works in a combined anticipation/deadline discipline; when I/O requests "expire" the scheduler ceases anticipation and gets into deadline mode. When measuring throughput we inadvertently assess the efficiency of disk-scheduling, prefetch as well as caching mechanisms combined. Hence, the exclusive use of throughput to determine the effectiveness of I/O scheduling disciplines might not provide much information as far as deceptive idleness is concerned. Nevertheless, we should ensure that the employed scheduling policies do not deteriorate throughput as this would certainly question the value of the policies at hand.

Through experimentation with *AS*-based schedulers, we established that all deliver comparable throughput rates, for all workloads in question. Minor throughput deviations among them are not the product of the heuristics used to set the anticipation interval but rather of the way the kernel dynamically triggers prefetch, exploits caches and/or follows the *Deadline* discipline for block-devices. Figures 18, 19, 20 and 21 depict the throughput rates attained by *Linux* operating under four of the workloads described in Section 4.3. The scheduling policies presented here are the *First Come First Serve (Noop)*, the *Deadline*, the *Complete Fair Queue* and the *Anticipation* configured with four different setups: the *95%-heuristic*, *LAS* with the fixed anticipation period, and the two *WAS(Cold-Boot)* and *WAS(Andrew's-like)* configurations. Overall,

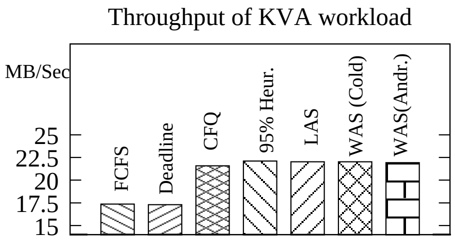

Figure 18: Throughput delivered by the *Linux* kernel *v.2.6.23* when experimenting with the *KVA* workload and different schedulers

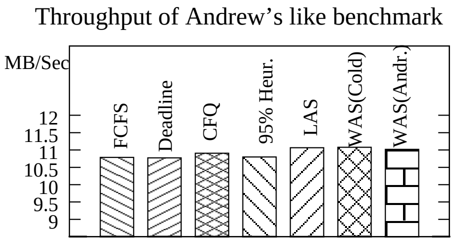

Figure 19: Throughput delivered by the *Linux* kernel *v.2.6.23* when experimenting with the *Andrew's-like* workload and different schedulers

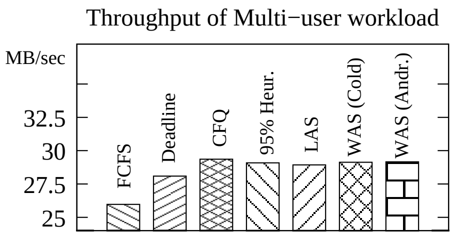

Figure 20: Throughput delivered by the *Linux* kernel *v.2.6.23* when experimenting with the Multi-User workload and different schedulers

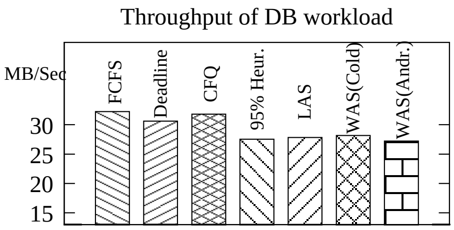

Figure 21: Throughput delivered by the *Linux* kernel *v.2.6.23* when experimenting with the DB workload and different schedulers

AS-based schedulers provide higher throughput rates for all workloads we worked with, with the only notable exception being that of the DB workload where long sequential reads are better served by traditional disk-scheduling disciplines. For the majority of the workloads where the *AS*-based schedulers fare best, it becomes a challenge to expose the benefits of anticipation exclusively via the reading of throughput rates. Here, there is a real need to offer additional measurements to more accurately quantify performance and better ascertain gains obtained in dealing with deceptive idleness. To this end, we introduce the metrics listed in Table 1.

Metric	Symbol	Description
Anticipated Requests	A	Number of times the anticipation scheduler has been triggered.
Successful Anticipations	A_S	Number of times an anticipated request occurred.
Failed Anticipations	A_F	Number of times an anticipation failed.
Anticipation Time	T	Time the anticipation scheduler spent in anticipation.
Successful Anticipation Time	T_S	Time the anticipation scheduler spent in successful anticipations.
Failed Anticipation Time	T_F	Time the anticipation scheduler spent in failed anticipations.

Table 1: *Anticipation Scheduling (AS)* Performance Metrics

From all the requests that the scheduler anticipates on (A), some do eventuate and others do not. The numbers of successful and failed anticipations are A_S and A_F respectively. The entire time that the disk is kept idle during anticipation is T. The time spent that resulted in successful anticipations is denoted T_S, while the penalty time for keeping the disk idle without success is T_F. Of course, equations $A = A_S + A_F$ and $T = T_S + T_F$ hold. The most notable of the metrics in Table 1 are A_S and T_F as A_S is a reliable way to measure the effectiveness of *AS*-schedulers and T_F can successfully capture the penalty time. We use the fraction T_F/A_S to measure the average penalty sustained for every successful anticipation; this fraction reflects the accuracy with which *AS*-based disciplines work. Apparently, the lower the fraction T_F/A_S becomes the better we deal with deceptive idleness. Furthermore, since the penalty we pay for a successful anticipation essentially corresponds to the time the hard-disk stays idle, T_F/A_S offers a fruitful way to measure application responsiveness and disk utilization overhead. Below, we provide a few guidelines on how to interpret the obtained experimental results.

1. The current *Linux* kernel with its constant anticipation period serves as a point of reference.
2. Our prime goal is to decrease the penalty time T_F while performing equally well in terms of successful anticipations with the current *Linux* kernel. Our secondary objective is to increase the number of successful anticipations while maintaining T_F constant when possible.
3. As selective workloads produce scattered I/O requests throughout the disk, they have little to gain from *AS*-based scheduling. The goal here is to diminish the penalty time which calls for the use of the shortest possible anticipation intervals.

In what follows, we outline the results produced by running the four *AS*-scheduler variations on a number of workloads. As the experimentation occurred on a real system, all experiments were repeated many times to warrant confidence in the results presented.

4.3 Experimental Results using Diverse Workloads

•**The *KVA*-Synthetic Workload:** exposes the weak points of the current *LAS* scheduler and reveals the enhanced quality in terms of timing penalties and missed anticipations that *WAS* offers. *KVA* comprises of two phases: in its first phase, a large file with random data is created. The file is large enough so that it does not fit in main memory entirely—more than $2\,GB$ for our system setup. In the second phase of *KVA*, the file is read by two processes. Each process reads in $4\,KB$ blocks from different areas of the file to avoid cache hits. These processes place successive requests that are $128\,KB$ apart, in order to disable the prefetch mechanism and force the requests to certainly go to the block device for service. In addition, one out of every eight requests is not issued immediately as the calling process stalls for more than $6\,ms$. This waiting period is just enough to make the anticipation interval expire and artificially introduce a number of anticipation fails in the *Static* $6\,ms$ *LAS* configuration.

Table 2 presents the results under this synthetic workload. The second column presents the results of the *95%-heuristic* described in [ID01]. The third column holds the observed values for the *Static* 6 ms *LAS* scheduler. The fourth and fifth columns display the values of *WAS* using configurations derived with the help of the *Cold Boot* and *Andrew's-like* workloads. The number of failed anticipations drops by approximately 70%; from 5,438 in the *Static* 6 ms case to 1,724 and 1,893 for the *WAS* schedulers. Reduced anticipation failures lessen the penalty time by around 60% from the penalty sustained during the constant 6 ms operation. The *95%-heuristic* performs better than the *Static* 6 ms as it sustains less penalty time and also anticipates more requests. However, the average penalty for a successful anticipation of the *95%-heuristic* is greater compared to either of our implementations. Moreover, in the *KVA*-workload the *WAS* schedulers, apart from avoiding anticipation fails, they also introduce new anticipation successes. This is possible by assigning longer anticipation periods to promising classes in the classification scheme. In summary, the much reduced values in the T_F/A_S underlines overall enhanced handling of the deceptive idleness as far as the *WAS*

Anticipation	*95%-Heuristic*	*Static* 6 *ms*	*WAS (Cold Boot)*	*WAS (Andrew's-like)*
A	84,483	83,851	85,341	84,407
A_S	81,165	78,413	83,448	82,683
A_F	3,318	5,438	1,893	1,724
T	24,817 ms	37,487 ms	21,316 ms	17,994 ms
T_S	5,711 ms	4,859 ms	7,846 ms	4,814 ms
T_F	19,106 ms	32,628 ms	13,470 ms	13,180 ms
T_F/A_S	0.23	0.41	**0.16**	**0.15**

Table 2: *KVA*-benchmark results

schedulers is concerned.

•**The** *Cold Boot* **Workload:** involves the initiation of services at boot time that include *acpi, system log daemons, sshd, cupsys, dbus and kde.* Many of these services are launched concurrently in distinct batches. For instance *acpid, powernowd.early, sysklogd* and *xserver-xorg-input-wacom* are grouped together. As soon as the *X-server* starts, *KDE* automatically launches a number of applications that entails *xmms, Konsole*, ten different files are opened with *Kwrite*, eight instances of the *Konqueror* browser are started pointing at different locations of the file-system and lastly *Open Office(v.2.3)* is used to open a text document. The simultaneous launch of the above processes forces successive block requests and as a result the anticipatory scheduling plays a significant role.

Table 3 shows the measurements obtained for *Cold Boot* using the four scheduler configurations. The

Anticipation	*95%-Heuristic*	*Static* 6 *ms*	*WAS(Cold Boot)*	*WAS(Andrew's-like)*
A	2,752	2,749	3,540	3,633
A_S	2,558	2,501	3,256	3,333
A_F	194	248	284	300
T	3,583 ms	3,641 ms	5,495 ms	6,076 ms
T_S	2,383 ms	2,153 ms	4,054 ms	4,539 ms
T_F	1,200 ms	1,488 ms	1,441 ms	1,537 ms
T_F/A_S	0.469	0.595	**0.442**	**0.461**

Table 3: Results of the *Cold Boot* Benchmark

number of anticipations increased for the *WAS*-schedulers compared to those of *Static* 6ms *LAS*. The reason behind this is that a good number of adopted classes are promising for anticipation and the *WAS*-schedulers decide to assign longer anticipation periods to them. This decision led the *WAS(Cold Boot)*-scheduler in increasing T_S, from 2,153 ms in *Static* 6ms *LAS* to 4,054 ms –a 47% increase– and the *WAS(Andrew's-like)*-scheduler in increasing T_S to 4,539 ms –a 53% increase. The two *WAS*-schedulers show similar results with the one using the *Cold Boot* configuration having the advantage. Clearly, this happens due to the fact that the workload examined was also used for the production of the classification scheme for *WAS(Cold Boot)*. As depicted by the total number of anticipations A and the amount of time spent T, the *95%-heuristic* provides similar results to the *Static* 6ms *LAS*, yet, when it comes to failed anticipations A_F and penalty time T_F the *95%-heuristic* shows enhanced performance (19% decrease in penalty time over *LAS*). Although the *95%* rule is an improvement over *LAS*, the T_F/A_S factor ranks the *WAS* schedulers higher.

•**The** *Andrew's-like* **Benchmark:** has been the standard benchmark for measuring file-system performance for many years [HKM+88]. The workload involves the creation of both files and directories, compilations, and searching through the content of the files. In our variation termed *Andrew's-like*, we used 8 processes which start concurrently with each one sequentially executing 150 *Andrew's* benchmarks. Table 4 shows the results of our experimentation. As is the case of *Cold Boot*, *Andrew's-like* workload also incorporates short

Anticipation	*95% Heuristic*	*Static* 6 *ms*	*WAS(Cold Boot)*	*WAS(Andrew's-like)*
A	23,790	40,389	39,439	39,731
A_S	15,138	32,065	31,593	31,934
A_F	8,652	8,324	7,846	7,797
T	48,510 ms	128,801 ms	126,774 ms	129,389 ms
T_S	20,377 ms	78,857 ms	77,936 ms	80,185 ms
T_F	28,133 ms	49,944 ms	48,838 ms	49,204 ms
T_F/A_S	1.858	1.557	**1.545**	**1.540**

Table 4: Results with an *Andrew's-like* benchmark

access patterns from different processes. Responding to such access patterns, the *WAS* schedulers avoid waiting for I/Os that are less promising and help lower the number of anticipations by 2–6%. The failed anticipations A_F dropped from 8,324 in the *Static* to 7,846–7,797 in the *WAS* settings which represents a sizable reduction. Consequently, the *WAS*-schedulers managed to attain a smaller penalty time per successful anticipation when compared with their *Static* counterpart. As expected, the *WAS(Andrew's-like)*-scheduler achieves a higher number of successful anticipations A_S than its *WAS(Cold Boot)* counterpart as it obtains its calibration with the help of the *Andrew's-like* workload. In this workload, the *95%-heuristic* cannot be directly compared with any of the other schedulers. The penalty time is only half the one we get with *Static* 6 *ms*, but the number of anticipations is less by 40%. The large number of failures in *95% heuristic* render the anticipation policy ineffective and the scheduler works mostly in deadline mode. Indeed, the penalty per success of the *95% heuristic* performs clearly the poorest of the four schedulers.

•**The** *Kernel Compilation* **Workload:** is not a typical workload for numerous small files are accessed by different processes. This workload would nominally call for no anticipation as most of the files are read in their entirety in just one I/O. Even when the files are larger than the block size, the read ahead policy of the disk-scheduler is most probably able to prefetch the remaining blocks of the file. This workload is generated by calling *make* on the kernel source tree with the "`-j`" option, in order to spawn multiple processes for concurrent compilation of the source. In our case, we use four (4) processes, which in turn start children processes for the compilation of different parts of the kernel source tree. This workload offers a good example where *WAS* is helpful as it places requests to classes that are not given long anticipation intervals. This minimizes the penalty times involved.

Table 5 shows the results of experimenting with the *kernel compile* workload. Compared with *Static LAS*

Anticipation	*95%-Heuristic*	*Static* 6 *ms*	*WAS(Cold Boot)*	*WAS(Andrew's-like)*
A	4,183	5,654	3,792	3,544
A_S	3,684	5,083	3,406	3,171
A_F	499	571	386	373
T	10,075 ms	10,411 ms	6,701 ms	6,567 ms
T_S	7,616 ms	6,985 ms	4,549 ms	4,673 ms
T_F	2,459 ms	3,426 ms	2,152 ms	1,894 ms
T_F/A_S	0.66	0.67	**0.63**	**0.59**

Table 5: Results from the *kernel compile* workload

case, where the penalty time T_F is 3,426 ms, the *WAS*-schedulers decrease T_F significantly: *WAS(Cold Boot)* lowers T_F to 2,152 ms by 37%, while *WAS(Andrew's-like)* drops T_F even more to 1,894 ms by 45%. Obviously, *WAS(Andrew's-like)* configuration uses a more suitable classification scheme for the *kernel compile* as the *Andrew's-like* and this workload share similar processes such as *gcc, ld*, etc. In general, *WAS* appears to be able to identify many I/O requests as not good candidates for anticipation. Indeed, out of the 10,075 ms spent in request anticipation, 32% of time is lost. *WAS*-schedulers reduce their respective total anticipation times T to 6,701 ms and 6,567 ms, for they identify anticipation is not beneficial in this workload. The *95%-heuristic* is unable to "recognize" the characteristics of this workload. The T_F/A_S factor shows that *WAS(Andrew's-like)* configuration does best in dealing with the deceptive idleness of this workload.

•**The** Bonnie++ **Workload:** consists of many sequential reads and writes confined in $2GB$ of disk space. Contemporary multi-threaded information retrieval systems or database engines are systems that may yield such I/O behavior. In this sense, the workload produced by Bonnie++ benchmark can directly benefit from anticipation. In Table 6, we present the results of our testing with the four schedulers. We observe that the *Static* scheduler is running with a failure rate of 19%. The *WAS*-schedulers improve the time spent on

Anticipation	*95%-Heuristic*	*Static* 6 ms	*WAS(Cold Boot)*	*WAS(Andrew's-like)*
A	23,790	40,923	39,939	40,731
A_S	15,138	33,080	32,093	32,934
A_F	8,652	7,843	7,846	7,797
T	48,510 ms	99,258 ms	124,774 ms	126,389 ms
T_S	20,377 ms	52,200 ms	77,936 ms	80,185 ms
T_F	28,133 ms	47,058 ms	46,838 ms	46,204 ms
T_F/A_S	1.85	1.42	**1.45**	**1.40**

Table 6: Results of testing with Bonnie++ benchmark

successful anticipations T_S by approximately 50%. This is because *WAS* invests in stalling certain process classes that appear promising. Moreover, the penalty time T_F is unaffected when compared with the *Linux* default disk scheduler. The *95%-heuristic* is unable to accomplish the same results as the *Static* scheduler and works in deadline mode most of the time hence the reduced anticipations A.

•**The** Multi-User **Workload:** tries to emulate the I/O activity produced in a multi-user environment [SS91]. To this end, eight processes are executed simultaneously placing 3000 I/O requests each of which 60% of are reads and 40% writes. Every read request fetches from disk $16KB$ of data whereas with each write $4KB$ of data are stored. Each process operates on its own workspace containing 10,000 files. Of those 10,000 files 7,500 have size between $128KB$ and $256KB$ and the rest 2,500 files are sized between $256KB$ and $384KB$. To the extend the file-system allows it, all the files are clustered together according to the workspace they belong to. This is achieved by sequentially creating each and every workspace in a de-fragmented disk partition. Consequently, anticipation on a process will result in a request that is in the vicinity of its workspace. Additionally, we have enhanced this workload by inserting a random delay after each request leading to some anticipation failures.

Table 7 shows the experimentation results with the four schedulers. The *95%-heuristic* comes one step behind the other three configurations. Its attempts for anticipation are limited when compared to the *Static* configuration. The two *WAS*-schedulers show improved T_F/A_S values. They both sustain lower penalty time but the successful anticipations are barely decreased compared to the *Static* configuration. The overall penalty per successful anticipation criterion renders *WAS(Andrew's-like)* the best calibration choice for this frequently-encountered workload environment.

•**The** DB **Workload:** tries to create traces that resemble those generated by a database engine [SS91]. In

Anticipation	*95%-Heuristic*	*Static* 6 *ms*	*WAS(Cold Boot)*	*WAS(Andrew's-like)*
A	957	1,980	1,489	1,522
A_S	385	791	765	747
A_F	572	1,189	724	775
T	6,854 ms	10,3843 ms	7,830 ms	8,279 ms
T_S	2,332 ms	3,249 ms	2,692 ms	2,849 ms
T_F	4,522 ms	7,134 ms	5,138 ms	5,430 ms
T_F/A_S	11.74	9.02	**6.71**	**7.27**

Table 7: Results of experimenting with the Multi-User workload

this respect, DB features three types of files: large files from where all processes read data, one application log per process where each process records its actions and finally, one database log file which is used as a universal log by all processes. In our implementation, we used one large file with size of $200MB$, the universal log is set to $100MB$ and there are 5 processes each of which uses a $50MB$ application log.

Table 8 shows the results of our experimentation with the four configurations. In DB, every process repeatedly reads parts from the data file and then writes into its two log files. Evidently any attempt to

Anticipation	*95%-Heuristic*	*Static* 6 *ms*	*WAS(Cold Boot)*	*WAS(Andrew's-like)*
A	2,815	2,559	7,801	7,583
A_S	1,985	1,931	5,938	5,772
A_F	830	628	1,863	1,811
T	7,631 ms	9,284 ms	25,479 ms	26,530 ms
T_S	3,395 ms	5,516 ms	16,823 ms	18,383 ms
T_F	4,236 ms	3,768 ms	8,656 ms	8,147 ms
T_F/A_S	2.13	1.95	**1.45**	**1.41**

Table 8: Results of experimenting with the DB workload

anticipate an I/O request should "take into consideration" whether another process will soon place a nearby request. The *95%-heuristic* and *LAS* do not take this risk and do not conduct enough anticipations. On the other hand the *WAS* configurations follow a different approach. They manage to conduct more anticipations than *LAS* and *95%-heuristic* by assigning shorter anticipation periods (in the average about 4.5 ms per anticipation) and thus serve more requests before they start expire. The later in turn allowed *WAS* to stay in deadline mode for a greater period of time and anticipate more requests. The success of our configurations is depicted by a reduction of 28% in T_F/A_S if compared with the standard *Static* 6 *ms LAS*.

•**The** Scientific **Workload:** tries to mimic block-device access behavior of scientific applications [SS91]. We enhanced this workload in a way that involves concurrency and would thus call for anticipation. Five processes post read and write I/O requests in ratio 6 to 4 with a variable idle period in between to simulate CPU processing bursts. These requests are equally distributed to three types of files large($500MB$) medium ($100MB$)and small ($10MB$). The chunks of data accessed are either $512KB$ or $32KB$ long. Although we used 1 large file 3 medium and 10 small ones the I/O requests are still scattered throughout the disk.

Table 9 shows the results of our experimentation with the four schedulers on the Scientific workload. Scattered requests is a feature of scientific applications as they tend to exactly "know" where to fetch data from, read it in one seek and finally carry out the requisite processing. $\widehat{\text{W}}$AS is able to identify those process classes that provide high prospect for anticipation. As a result the successful anticipation attempts are slightly increased when compared to *LAS* and the *95%-heuristic* configurations. In addition, the failed

Anticipation	*95%-Heuristic*	*Static 6 ms*	*WAS(Cold Boot)*	*WAS(Andrew's-like)*
A	1,243	1,242	1,213	1,160
A_S	840	856	900	846
A_F	403	386	313	314
T	8,132 ms	10,744 ms	9,905 ms	8,085 ms
T_S	5,446 ms	8,428 ms	7,820 ms	5,958 ms
T_F	2,686 ms	2,316 ms	2,085 ms	2,127 ms
T_F/A_S	3.19	2.70	**2.31**	**2.51**

Table 9: Results of experimenting with the Scientific workload

anticipations along with the penalty time (T_F) decrease. This 18% decrease in failed anticipations of *WAS* over *LAS* is depicted by T_F/A_S.

4.4 Disabling the Prefetch Mechanism of the File-System

We have experimented with all the workloads and present here results obtained while the prefetching mechanism of the file-system was disabled. Our goal has been to better understand and quantify the effect of read-ahead when it comes to disk-scheduling. Although we have experimented with the entire range of the workloads used, for brevity we only report results from the *KVA*, *Andrew's-like*, Multi-User and DB benchmarks in Tables 10, 11, 12 and 13 respectively.

Anticipation	*95%-Heuristic*	*Static 6 ms*	*WAS (Cold Boot)*	*WAS (Andrew's-like)*
A	86,465	88,411	85,958	90,338
A_S	81,128	83,043	83,386	87,653
A_F	5,337	5,368	2,572	2,685
T	25,375 ms	37,177 ms	20,307 ms	19,758 ms
T_S	5,137 ms	4,969 ms	4,867 ms	4,924 ms
T_F	20,238 ms	32,208 ms	15,440 ms	14,834 ms
T_F/A_S	0.24	0.38	0.18	0.16
Throughput	22,851KB/sec	22,561KB/sec	22,851KB/sec	22,657KB/sec

Table 10: *KVA*-benchmark results with prefetch disabled

Anticipation	*95% Heuristic*	*Static 6 ms*	*WAS(Cold Boot)*	*WAS(Andrew's-like)*
A	56,745	77,005	78,121	76,756
A_S	46,174	68,235	70,246	68,997
A_F	10,571	8,770	7,875	7,759
T	69,762 ms	149,971 ms	158,774 ms	152,912 ms
T_S	35,923 ms	97,351 ms	107,087 ms	104,711 ms
T_F	33,839 ms	52,620 ms	51,687 ms	48,201 ms
T_F/A_S	0.73	0.77	0.74	0.7
Throughput	8,014KB/sec	9,574KB/sec	9,563KB/sec	9,540KB/sec

Table 11: *Andrew's-like* benchmark results with prefetch disabled

In all cases – except *KVA* – we see that disabling prefetch significantly increases the anticipation attempts conducted by the scheduler. The reason for this is that since read-ahead is disabled a lot of anticipations will be successful given the fact that most processes place successive I/O requests for successive disk sectors. Both prefetch and anticipatory scheduling are mechanisms to combat deceptive idleness: On the one hand prefetch chooses to read some blocks ahead of the requested in hope that the same process will ask for successive sectors; on the other hand anticipatory scheduling chooses to stall the disk in hope that the just serviced process will soon request some of the blocks the hard disk head is over, that is sectors that follow the just serviced request. Therefore, when one of the two mechanisms, prefetch or anticipation, is absent

Anticipation	*95%-Heuristic*	*Static 6 ms*	*WAS(Cold Boot)*	*WAS(Andrew's-like)*
A	51,052	55,335	54,852	54,535
A_S	43,628	45,789	45,872	45,842
A_F	7,424	9,546	8,980	8,693
T	92,308 ms	100,049 ms	92,763 ms	93,849 ms
T_S	43,121 ms	42,773 ms	41,863 ms	42,119 ms
T_F	49,187 ms	57,276 ms	50,900 ms	51,730 ms
T_F/A_S	1.12	1.25	1.10	1.12
Throughput	26,877KB/sec	26,812KB/sec	26,650KB/sec	27,009KB/sec

Table 12: Multi-User workload results prefetch disabled

Anticipation	*95%-Heuristic*	*Static 6 ms*	*WAS(Cold Boot)*	*WAS(Andrew's-like)*
A	51,377	56,755	55,961	56,514
A_S	50,238	54,880	54,086	54,624
A_F	1,139	1,875	1,875	1,890
T	28,356 ms	34,795 ms	43,273 ms	37,304 ms
T_S	17,483 ms	23,545 ms	32,486 ms	26,371 ms
T_F	10,873 ms	11,250 ms	10,787 ms	10,933 ms
T_F/A_S	0.22	0.20	0.19	0.20
Throughput	29,539KB/sec	27,749KB/sec	28,321KB/sec	28,032KB/sec

Table 13: DB workload results with prefetch disabled

the other one takes over and tries to fill the absence of the missing one. This is what we precisely saw in all the above experiments with the only exception being that of *KVA*. The latter was specifically build so as to by-pass kernels read-ahead (Section 4). Consequently, when we compare Tables 10 and 2 we do not see any considerable increase in the amount of the anticipations. The T_F/A_S factor overall points the *WAS* schedulers as very promising in terms of timing delays.

The last line in each of the above tables presents the average throughput of each workload when prefetch is disabled. When comparing these corresponding rates in Figures 18, 19, 20 and 21 we see a slight decrease. Although both anticipatory scheduling and prefetch target deceptive idleness, prefetch is more "aggressive" in its actions. Prefetch does not leave the disk idle at no time so when a process places a successive I/O it is served by data that are already prefetched in RAM.

4.5 *WAS* Operating on Slower Hardware

Our objective in testing *WAS* with a slower computing system is twofold: first, to establish the utility of our enhanced scheduler and second, to show that the produced classification scheme does reflect the hardware and system software under testing. We used a Pentium-M machine at $1.3\,GHz$ with $512\,MB$ of main memory and $30\,GB$ of disk running *Suse 9.1* patched with the 2.6.4 kernel using *WAS*. We produced two classification schemes based on the *Andrew's-like* benchmark and *Cold-Boot* –termed *Slow Andrew's* and *Slow Cold Boot*– as Figures 16 and 17 show. We can readily see from these two graphs that although slow and fast hardware yield similar spatial anticipation prospect functions their temporal counterparts are very different. In combination, the respective prospect functions for fast/old hardware do provide two distinct classification schemes.

We experimented with all workloads/configurations [6] and Tables 14 and 15 shows some of the results. The table also shows the corresponding results, should we use the standard *LAS*-scheduler. In all our experiments and in comparison with the $LAS\,6ms$ values, we see that the penalty time T_F values are reduced while the number of successful anticipation A_S attempts either improves a little or remains unchanged. The combined

[6] Shorter versions of workloads were produced to match the hardware capabilities.

effect of T_F and A_S renders our implementation as most promising.

	KVA			*Andrew's-like*		
Anticipation	*Static 6 ms*	*WAS(Cold Boot)*	*WAS(Andrew's)*	*Static 6 ms*	*WAS(Cold Boot)*	*WAS(Andrew's)*
A	4,717	4,638	4,711	17,535	16,521	17,100
A_S	4,251	4,517	4,577	12,224	11,910	12,502
A_F	466	121	134	5,311	4,611	4,598
T	4,851 ms	5,247 ms	5,293 ms	51,205 ms	46,604 ms	47,039 ms
T_S	2,055 ms	4,158 ms	4,147 ms	19,341 ms	19,542 ms	19,802 ms
T_F	2,796 ms	1,089 ms	1,146 ms	31,864 ms	27,062 ms	27,237 ms
T_F/A_S	0.66	**0.24**	**0.25**	2.60	**2.27**	**2.18**

Table 14: Results of *KVA* and *Andrew's-like* benchmarks on older hardware

	Kernel Compile			Bonnie++		
Anticipation	*Static 6 ms*	*WAS(Cold Boot)*	*WAS(Andrew's)*	*Static 6 ms*	*WAS(Cold Boot)*	*WAS(Andrew's)*
A	199	139	97	13,027	13,575	13,299
A_S	59	47	64	12,737	13,335	13,074
A_F	140	92	33	290	240	225
T	947 ms	464 ms	223 ms	6,857 ms	6,980 ms	6,899 ms
T_S	107 ms	34 ms	43 ms	5,117 ms	5,616 ms	5,549 ms
T_F	840 ms	430 ms	180 ms	1,740 ms	1,364 ms	1,350 ms
T_F/A_S	14.23	**9.15**	**2.81**	0.14	**0.10**	**0.10**

Table 15: Results of *Kernel Compile* and Bonnie++ benchmarks on older hardware

Lastly, we should point out that there is no benefit in directly comparing the results obtained with the slower hardware to those derived from the faster. This has to do with the fact that slower request service causes the I/Os to be scheduled differently. Moreover, the total number of anticipated requests is also related to the size of the main memory each system has.

4.6 Discussion of Results and Overheads

Table 16 summarizes the T_F/A_S measurements for both *LAS* and *WAS* schedulers across all the workloads used. From the two configurations we deploy for *WAS*, we depict for each case the one that better suits the workload in question. *WAS* offers time savings for successful anticipations for all workloads that in some

Workload	*KVA*	*Cold Boot*	*Andrew's*	*Kernel Comp*	Bonnie++	Multi-User	DB	Scientific
LAS	0.41 ms	0.595 ms	1.557 ms	0.67 ms	1.42 ms	9.02 ms	1.95 ms	2.70 ms
WAS	0.15 ms	0.442 ms	1.540 ms	0.59 ms	1.40 ms	6.71 ms	1.41 ms	2.31 ms

Table 16: Penalty per successful anticipation for each workload tested

workloads may halve the respective rates of the *Linux* standard disk-scheduler.

The overheads involved in *WAS* are rather minimal when compared to the milliseconds gained. In its regular operational mode, *WAS* does work in kernel and initially for each I/O updates the history of the process and then determines the anticipation interval that the process should impose on the block-devices. As the history is a short cyclic list with a pointer to its last record (Section 3.2), updating this structure requires no list traversal. Consequently, the pertinent overhead in carrying out history update is indeed minimal. When *WAS* has to determine the class a process belongs to all elements of the process history have to be traversed (Section 3.3). As soon as *WAS* determines the specific class, it then carries out an array look-up to find out the anticipation interval to be used. Should we consider that an ever increasing number of Instructions Per Second (IPS)[PH07] are performed by modern CPUs, the above in-kernel overheads are rather negligible especially if compared to milliseconds saved for each successful anticipation. Even when *WAS* enters its *observation phase*, it does call for minimum computational resources as this phase (Section 3.6) is equivalent to simply increasing an element in matrix A of Section 3.6 every time an anticipation is carried out. The *v.2.6.23* kernel offers no tools for measuring such extremely short time periods and any attempt of our own to instrument the code would certainly provide inaccurate measurements at this fine granularity level.

WAS user-space components may impose notable overheads. However both *Classification* and *Designation Subsystems* are meant to be used infrequently and only when the overall requirements of the computing system allow for the extra overhead. More specifically, *Classification Subsystem* is invoked only once during the system installation (Section 3.4) and the *Designation Subsystem* (Section 3.6) only once per workload. Taking into account that workloads with similar characteristics are often recurring[Dil96], overheads imposed by the two subsystems could be better handled by more effective use of available computing system resources. For instance in a server environment, workloads may recur every 24-hours and the type of the workloads may change every week or month. Here, *WAS* can rapidly adapt to incoming workloads by using existing scheduler configurations that characterize the anticipated I/O traffic through a *cron*-job. In this way, the system sustains no downtime. Only if no appropriate scheduler configuration exists, *WAS* may work harder to compile all the required statistics and yield viable process classes and assignments of anticipation intervals per process class.

5 Related Work

There have been numerous efforts to improve the performance of block devices and in particular disks by mostly conserving on the work that mechanical parts carry out [Fin86, Tan01]. Disciplines including the Shortest-Seek-latency-First (SSF), SCAN, Look and C-Look attempt to minimize the distance traveled by the disk arm. The careful placement of files on the disk based on their expected locality has been suggested in [MNN03] while the adaptive re-arrangement of data blocks on the surface of the disk plates [VC90, AS97] have been used as ways to enhance the I/O throughput of storage devices. In [RR03], a novel storage architecture subsystem is outlined in which more than one requests can be placed within the actual device at any given time providing chances for a more opportune disk scheduling. Data compression [CLRS01, Lin05] could be helpful in increasing the "effective" disk-transfer rates regardless of the choice made as far as the device scheduler is concerned.

In [ID01], it was shown that *AS* works well in conjunction with the *Apache* web-server with respect to I/O-intensive workloads, for file-system benchmarks, as well as for variations of the TPC-B benchmark. As *AS* combats the deceptive idleness of a process, it was shown that better facilitates workloads where multiple synchronous I/Os are present. In contrast, workloads of random and scattered I/Os may induce high penalties for the *AS*.

Modern kernels offer a range of policies for disk scheduling in order to accommodate for the different types of traffic a block-device may handle [J. 00, MNN03, Vah96]. Combining the characteristics of the block-device at hand and the expected workload, system administrators may select an appropriate scheduling policy. The *Linux* anticipatory disk-scheduler occasionally anticipates I/O requests merely using a fixed-length wait period [Lov05]. For the *FreeBSD* implementation of *AS*, the anticipation period is determined with statistics only related to the arrival times of the I/Os of a process [Iye01]. The stall period is set so that there is a 95% chance for successful anticipation. Although this heuristic has been shown to outperform traditional disk-scheduling policies, it has two limitations: firstly, it does not capture the spatial relationships of I/Os deemed essential in characterizing process behavior and secondly, the above estimation of the successful percentile is computationally intensive while working in the kernel space.

A number of alternative policies have also emerged in order to address requirements for specialized applications. For example, a wide range of environments with real-time needs have spurred the development of scheduling techniques for requests to either devices or systems based on deadlines [BB94, BSR88, CSKT91, AGM92]. A deadline-based scheduler accepts requests with a maximum wait time and tries to always fulfill the largest number of those processes present in its queue. Evidently, this is not always feasible leading to either aborted or resubmitted requests. In multimedia environments where quality-of-service requirements have to be observed at all times for the avoidance of jitter, the fair queue scheduling has been suggested [SV98, DKS89]. Here, the scheduler tends to many processes that require substantial service from disks in a fair way while preventing a subset of the competing processes from monopolizing the use of the devices in question.

6 Conclusions and Future Work

Effective disk scheduling is required to obtain not only good overall performance for block-devices but more importantly to continually attain good responsiveness for applications. The *Anticipatory Scheduling(AS)* works in this direction with a unique approach: as soon as a process I/O request has been served, it stalls the block-device for a period of time. Stalling is introduced in hope that a new request –from the just serviced process– for a nearby block will soon arrive; in this case, the follow-up request is efficiently serviced without "expensive" disk seeks. The rationale of *AS* is to generate fewer and/or less expensive seeks and in this way to contribute to shorter handling periods for I/Os.

In this paper, we build on the success of *AS* scheduling and seek to provide schedulers that use varying-length anticipation intervals. We argue that the types of workloads a file system is called to tend, in conjunction with the hardware/software computing system configurations should be taken into consideration to offer a more flexible and effective anticipation discipline. We propose an approach named *Workload-dependent Anticipation Scheduler (WAS)*, in which the anticipation interval is a function of both spatial and temporal characteristics exhibited by requesting I/Os. Our approach uses minimal auxiliary structures and negligible CPU-overheads in the kernel. Its operation depends on two subsystems –*Classification* and *Designation*– that run only infrequently in user-space. The role of the *Classification Subsystem* is to produce a viable classification scheme of process classes based on a *calibration* phase. The *Designation Subsystem* considers the set of finally adopted classes and assigns to them anticipation intervals of varying length by solving a linear optimization problem. On one hand, the classification scheme is highly dependent on physical and/or logical characteristics of the computing system at hand. On the other, major workload changes may trigger the optimization process of the *Designation Subsystem* that re-evaluates the assignment of anticipation intervals for each process class identified during the execution of the classification.

We have incorporated our *WAS* approach into the *Linux* kernel *v.2.6.23* and experimented with a wide range of workloads. During our experimentation, we placed emphasis in investigating the penalty time imposed by failed anticipations and the responsiveness of the system defined by the number of successful anticipations. Our testing involved I/O activity produced by the *Andrew's benchmark*, Bonnie++, *Cold Boot*, the compilation of the *Linux* kernel as well as the synthetic *KVA*, Multi-User, DB and Scientific workloads. Our results show an overall reduction in penalty time while the numbers of successful anticipations remain at the same or higher levels to those attained by the conventional *LAS*-scheduler.

It is worth pointing out that both *Classification* and *Designation* need to be invoked only on major workload changes and/or hardware updates. Here, *WAS* has to get adapted to better serve the new I/O patterns. Our choice for placing these two subsystems in the user-space enables them to function online without having special requirements in place such as to either recompile the kernel or reboot the system. We believe this choice is appealing as system administrators may "load" in the kernel a classification scheme and corresponding assignments (i.e., scheduler configurations) of their own choice. For instance, as the workload of a workstation may change over regular periods of time, the administrator may employ a *cron*-job to re-designate *WAS* classes of I/Os and respective anticipation interval assignments on the fly. In the future, we intend to develop mechanisms that would allow for the unsupervised execution of the *classification* phase. In this context, we plan to investigate the effectiveness of cluster analysis in creating improved classification schemes of I/O requests. We will also examine sampling techniques in order to render the binding of anticipation intervals to classes more CPU-lightweight as far as the linear optimization is concerned.

Acknowledgment: We are very grateful to the anonymous reviewers for their valuable comments that helped us significantly improve our work and its presentation.

References

[AGM92] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems*, 17(3), 1992.

[AS95] S. Akyurek and K. Salem. Adaptive Block Rearrangement. *ACM Transactions on Computer Systems*, 13(2):89–121, 1995.

[AS97] S. Akyurek and K. Salem. Adaptive Block Rearrangement Under UNIX. *Software Practice and Experience*, 27(1):1–23, 1997.

[BB94] A. Bestavros and S. Braoudakis. Timeliness via Speculation for Real–Time Databases. In *Proceedings of the IEEE Real–Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[BBG+99] J.L. Bruno, J.C. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk Scheduling with Quality of Service Guarantees. In *Proceedings of the 1999 IEEE Int. Conf. on Multimedia Computing and Systems*, pages 400–405, Florence, Italy, June 1999.

[BC05] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reily, Sebastopol, CA, third edition, December 2005.

[BSR88] S. Biyabani, J. Stankovic, and K. Ramamritham. The Integration of Deadline and Criticalness in Hard Real–Time Scheduling. In *Proceedings of the Real–Time Systems Symposium*, pages 152–160, Huntsville, AL, December 1988.

[Car07] C.D. Carothers. Lecture Notes on CPU History. Rensselaer Polytechnic Institute, Troy, NY, www.cs.rpi.edu/~chrisc/COURSES/CSCI-4250 /SPRING-2004/slides/cpu.pdf, 2007.

[CDB04] Z. Chen, A. Delis, and H.L. Bertoni. Radio-wave Propagation Prediction Using Ray-tracing Techniques on a Network of Workstations (NOW). *Journal of Parallel and Distributed Computing*, 64(10):1127–1156, October 2004.

[CH81] R. Carr and J. Hennessy. WSCLOCK-A Simple and Effective Algorithm for Virtual Memory Management. In *Proceedings of the 8th ACM Symposium on Operating System Principles*, Pacific Grove, CA, December 1981.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press–McGraw Hill, Cambridge, MA, second edition, September 2001.

[Cok08] R. Coker. Bonnie++ file system benchmark. URL: http://www.coker.com.au/bonnie++/, 2008.

[CSKT91] S. Chen, J.A. Stankovic, J.F. Kurose, and D.F. Towsley. Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems. *Real-Time Systems*, 3(3):307–336, 1991.

[Den68] P.J. Denning. The Working Set Model of Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[Dil96] J. Dilley. Web Server Workload Characterization. Technical report, Hewlett-Packard Laboratories, 18 December 1996. URL http://www.hpl.hp.com/techreports/96.

[DKS89] A.J. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the ACM-SIGCOMM Symposium*, pages 1–12, Austin, TX, 1989.

[Fin86] R. A. Finkel. *An Operating Systems Vade Mecum*. Prentice, Englewood Cliffs, NJ, 1986.

[FNS95] C. Faloutsos, R.T. Ng, and T.K. Sellis. Flexible and Adaptable Buffer Management Techniques for Database Management Systems. *IEEE Transasctions on Computers*, 44(4):546–560, 1995.

[HKM+88] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[HLC90] J. Haritsa, M. Livny, and M. Carey. On Being Optimistic about Real–Time Constraints. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, Nashville, TN, 1990.

[ID01] S. Iyer and P. Druschel. Anticipatory Scheduling: a Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, New York, NY, 2001.

[Iye01] S. Iyer. *FreeBSD* Patch for Anticipatory Scheduler. http://www.cs.rice.edu/ ssiyer/r/antsched/antsched/, 2001.

[J. 00] J. Mauro and R. McDougall. *Solaris Internals: Core Kernel Architecture*. Sun Microsystems Press, Mountain View, CA, 2000.

[KM05] D. Kontos and V. Megalooikonomou. Fast and Effective Characterization for Classification and Similarity Searches of 2D and 3D spatial Region Data. *Pattern Recognition*, 38(11):1831–1846, 2005.

[Lin05] Linux Weekly News. http://www.lwn.net, 2005.

[Lov05] R. Love. *Linux Kernel Development*. Developer's Library Sams Publishing/Novel, second edition, 2005.

[LZGS84] E.D. Lazowska, J. Zahorian, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance*. Prentice Hall, Englewood Cliffs, NJ, 1984.

[Man84] J. Mandel. *The Statistical Analysis of Experimental Data*. Dover Publications, Inc., Mineola, NY, 1984.

[MMF84] S.J. Leffler M.K. McKusick, W.N. Joy and R.S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[MNN03] M.K. McKusick and G.V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, New York, NY, July 2003.

[OBRS94] B. Özden, A. Biliris, R. Rastogi, and A. Silberschatz. A Low–Cost Storage Server for Movie on Demand Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, September 1994.

[PH07] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. Elsevier Science & Technology, San Francisco, CA, USA, 3rd revised edition, 2007.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization Algorithms and Complexity*. Dover Publications, Inc, Mineola, NY, 1982.

[RF77] M. Ruschitzka and R.S. Farby. A Unifying Approach to Scheduling. *Communications of the ACM*, 20(7):469–478, 1977.

[RO91] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th ACM Symposium in Operating System Principles*, volume 25, pages 1–15, Pacific Grove, CA, October 1991.

[RR03] A. Riska and E. Riedel. It's not Fair–Evaluating Efficient Disk Scheduling. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems (MASCOTS)*, Orlando, FL, October 2003.

[SGG03] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts*. J. Wiley & Sons, New York, NY, sixth edition, 2003.

[SS91] M. Seltzer and M. Stonebraker. Read Optimized File System Designs: a Performance Evaluation. In *Proceedings of the 7th IEEE International Conference on Data Engineering*, Kobe, Japan, 1991.

[SV98] D. Stiliadis and A. Varma. Efficient Fair Queueing Algorithms for Packet-Switched Networks. *IEEE/ACM Transactions on Networking*, 6(2):175–185, April 1998.

[SV06] K. Stoupa and A. Vakali. QoS-Oriented Negotiation in Disk Subsystems. *Data and Knowledge Engineering Journal*, 58(2):107–128, 2006.

[Tan01] A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Upper Saddle River, NJ, 2nd edition, 2001.

[TK05] S. Theodoridis and K. Koutroumbas. *Pattern Recognition.* Academic Press, New York, NY, 3rd, chapters 2-4 edition, 2005.

[Tra08] Kernel Traffic. http://www.kerneltraffic.org, 2008.

[Vah96] U. Vahalia. *UNIX Internals – The New Frontiers.* Prentice Hall, Inc., Upper Saddle River, NJ, 1996.

[VC90] P. Vongsathorn and S. Carson. A System for Adaptive Disk Rearrangement. *Software Practice and Exprience*, 20(3):225–242, March 1990.

[WGP94] B.L. Worthington, G.R. Ganger, and Y.N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference*, pages 241–251, Nashville, TN, May 1994.