Idleness is not sloth

Richard Golding, Peter Bosch, and John Wilkes

idleness,
storage systems,
detecting idle periods,
predicting idle periods

Many people have observed that computer systems spend much of their time idle, and various schemes have been proposed to use this idle time productively. We have used this approach to improve overall performance in storage systems. The most common approach is to off-load activity from busy periods to less-busy ones in order to improve system responsiveness. In addition, speculative work can be performed in idle periods in the hope that it will be needed later at times of higher utilization, or a non-renewable resource power can be conserved by disabling unused resources during idle periods.

Much existing work in scheduling for idle periods uses ad hoc mechanisms. The work presented here includes a taxonomy of idle-time detection and prediction algorithms that encompasses the prior approaches and also suggests a number of others. We identify metrics that can be used to evaluate all these idle-time detection algorithms and demonstrate the utility of both the metrics and the taxonomy by providing a quantitative evaluation.

This paper is a revised, extended version of a paper presented at the 1995 Winter Usenix conference.

1 Introduction

Resource usage in computer systems is often bursty: periods of high utilization alternate with periods when there is little external load. If work can be delayed from the busy periods to the less-busy ones, resource contention during the busy periods can be reduced, and perceived system performance can be improved. The low-utilization periods can also be exploited for other purposes—for example, eagerly performing work that might be needed in the future, or shutting down parts of a system to conserve power, reduce wear, or lower thermal load.

The three main contributions of the work described in this paper are: a taxonomy of mechanisms that can be used to detect and predict low-utilization (idle) periods, a survey of a wide range of such mechanisms being used on some sample problems drawn from our storage systems work, and an analysis of why the approach worked well. The taxonomy makes it easy to describe such mechanisms and invent new ones; the survey and analysis provide concrete suggestions for appropriate algorithm choices in a range of circumstances.

We call periods of sufficiently-low resource utilization *idle periods*. The definition of "sufficiently low" is application specific; we use the term "idle" even if the value is non-zero. During these times the system can execute an *idle task* without using resources that will affect time-critical work too much.

Our approach is to detect when idle periods occur, to predict how long they will last, and then to use this information to schedule the idle task. When a sufficiently-long idle period is predicted, the idle task begins running. The idle task executes until it completes, or until it is signalled to stop—typically when new foreground work arrives.

Figure 1 shows the overall structure of the framework of the idle detection system we have investigated. Foreground work requests arrive and are executed, requiring resources. Potentially useful idle tasks also consume the same resources. An *idleness detector* monitors the foreground-work arrivals and the state of the scheduler, and many detector algorithms use the recent history of these events to guide their predictions of idle time. The detector can also consider environmental factors such as time of day. The idleness detector gives its predictions to the actuator, which schedules the background idle work.

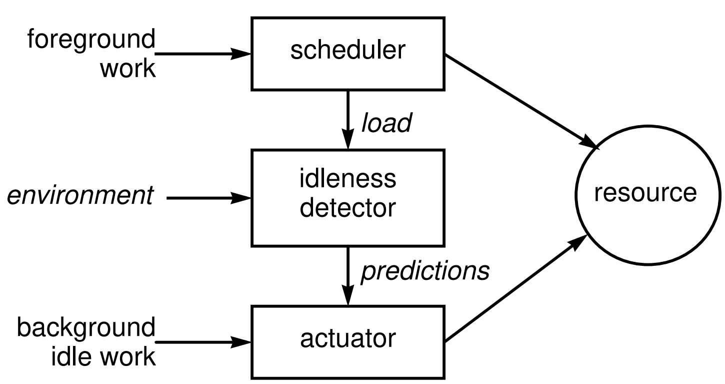

Figure 1: the idle-time processing framework.

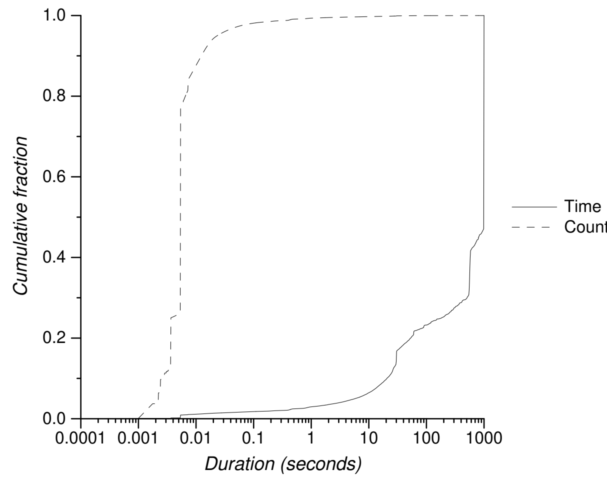

Figure 2: cumulative distribution of idle time as a function of idle period duration, for the cello-disk6 trace. The upper curve shows the fraction of the number of idle periods; the lower curve shows the fraction of total idle time.

The goal of the idleness detector is to make sufficiently good predictions that the net effect to the system of running the idle task is positive. The best predictions exploit all the idle time and make no mistaken predictions.

There are two basic ways to measure how good an idle-time processing system is. *External measures* quantify the interference between the idle task and an outside application, and the benefit from running the idle task. They use units such as additional foreground operation latency or power consumption. *Internal measures* are based solely on how accurate the detector's predictions are. The external measures are what really matter, but internal measures are easy to obtain and can be useful for guiding the choice of detection mechanism.

The rest of this paper is organized as follows. The rest of this section discusses some of the characteristics of the idle periods in storage systems we have investigated. The following section looks at several aspects of the problem of making good predictions about this idle time, including a discussion of how they can be measured. We then present an architecture and taxonomy for idleness detectors, use this taxonomy as a tool to generate a suite of idleness detectors, and evaluate their effectiveness under realistic, measured workloads. An analysis of the effectiveness of the taxonomy and an investigation of how one can choose an idleness detector follow. Some thoughts about opportunities for future work and our conclusions wrap up the paper.

1.1 The nature of idle time

In our research, we have analyzed traces of storage operations taken from a number of systems, including file systems, databases, and RAID arrays. We will be presenting results from traces taken from file system disks attached to HP-UX systems. Details on most of these traces can be found in our earlier paper on modeling disk systems [Ruemmler93].

Most idle periods in these traces are very short—on the order of milliseconds—but the bulk of idle time comes from a few very long periods. (Figure 2 shows one such distribution.) This means that

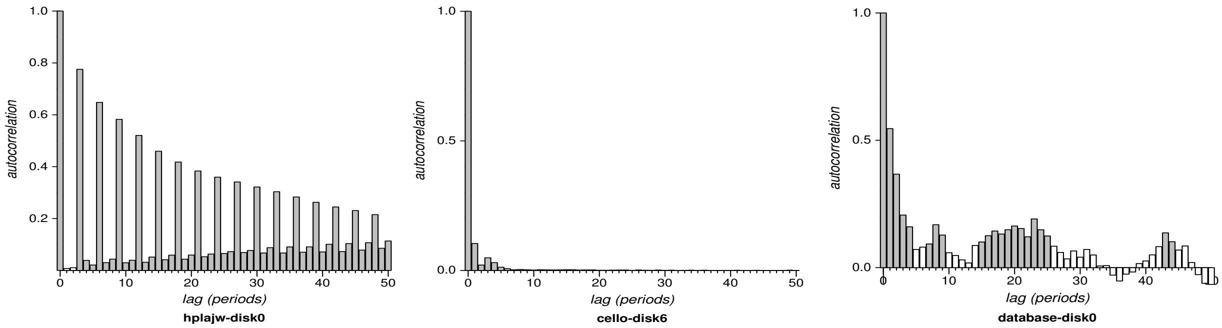

Figure 3: autocorrelation at different lags of the sequence of idle period duration, for three workloads. Dark bars indicate significant correlation at 95% confidence level.

a system that uses idle time can get most of the benefit by doing a good job of finding the long idle periods.

Idle periods also exhibit predictable patterns. To determine this, we computed the autocorrelation for the sequence of idle period lengths. Figure 3 shows the results: for most of the traces we investigated, how long the system stays idle is strongly related to how long it has recently been idle. The figure shows the three patterns we observed among the traces.

- hplajw-disk0: this trace was taken on a quiet personal workstation. Most of the idle periods were very long, and there was strong correlation between the length of one period and the lengths of the preceding periods.
- cello-disk6: this trace is from a disk holding the Usenet news partition on a time-sharing system. The length of time the system stayed idle depended only on the four previous periods, and there appeared to be little longer-term correlation.
- database-disk0: this disk held part of a large production database system that exhibited a transaction-processing-like workload. As with the cello-disk6 trace, the lengths of the previous few idle periods are strongly correlated with the length of the current one, but there are also long-term correlations at lags of more than forty periods.

Other researchers have indicated that file system traffic exhibits self-similar behavior [Gribble96]. In particular, they have shown that there is substantial long-term dependence between the durations of idle periods.

These results suggest that idleness is a predictable phenomenon: by observing recent behavior, and using the dependence between recent and future events, one can make good predictions about future behavior.

2 An overview of idle-time processing

Idle-time processing consists, first, of detecting an idle period, then using that knowledge to execute idle tasks to benefit overall system performance. In this section we explain what we mean by each of these things.

2.1 Detecting idle periods

Recall that the *idleness detector* monitors external work requests in order to find idle periods. More precisely, the detector's problem is to make a series of predictions, each of which identifies the *start time* and *duration* of an idle period. The detector cannot be late with a prediction—otherwise it isn't a prediction. A good prediction will neither start earlier than the actual start time (so there is no collision between idle processing and ordinary work) nor start much later (which would waste time the idle task could spend doing work); nor will the duration last beyond the end of the actual idle period.

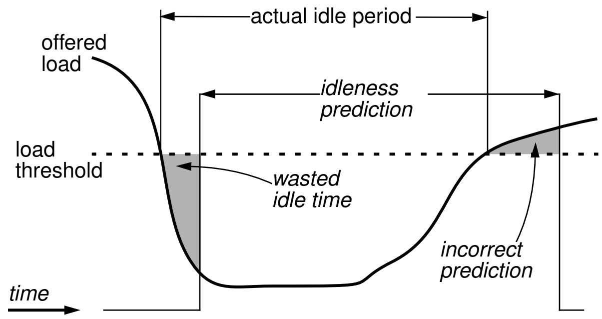

Figure 4: the output from a sample idle detector as the offered load changes.

As shown in Figure 4, the system initially is doing useful work, but then the offered external load decreases below a predetermined threshold. Some time after this happens, the detector makes a prediction about the idle period and signals the idle task to start. Later, as the load increases past the threshold, it signals the task to stop.

The time lag between the load dropping below the threshold and the detector signalling the start of the idle period represents a lost opportunity. This can also happen if the prediction is shorter than the actual idle period. On the other hand, the time lag between the load exceeding the threshold and the end of the prediction represents a possible actual cost because of interference with foreground activity.

2.2 Executing idle tasks

Idle tasks are run when the system is predicted to be idle. The idleness detector finds periods when the system will be idle, and starts and stops idle tasks appropriately.

The time line in Figure 5 illustrates a prototypical idle task in more detail.

Typically, the detector signals the idle task to start executing some time after the end of the last piece of foreground work, or when the load on the system drops below some threshold. The initial activity of the idle task may be to run an initialization step, such as determining which segments to clean in a log-structured file system. This is followed by one or more executions of the main idle-time step. Each step might take a different amount of time or require a different resource. Breaking the idle task up into finer-grained steps reduces delay after an interruption, and hence reduces the cost of a bad prediction.

Eventually, regular foreground work will again enter the system, or the system load will increase above a threshold. If the detector's prediction was accurate, the idle task will have completed execution. If not, the detector signals the idle task to stop, and the task interrupts its activity, if

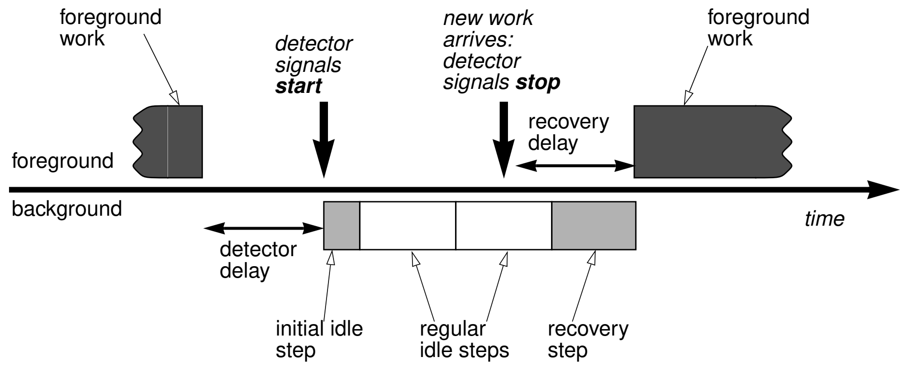

Figure 5: the typical sequence of events in idle-time processing.

possible. It may be necessary to execute a recovery step to bring the system back to normal operation—for example, a powered-down disk must be spun up, or a partially-completed update may need to be undone. Foreground work may be blocked or slowed during this period, so it is desirable to keep the recovery time as short as possible.

2.3 Costs and benefits of using idle time

To understand the benefits and costs of idle-time processing, we must first understand how idle tasks can help a system, and how they can hinder.

Measuring the improvement requires metrics for each idle task. These measures vary from one system or application to another. Even within one system, more than one measurement may be appropriate. Often these measures are not directly comparable, or may be subjective in nature. *External performance measures* quantify these effects.

For example:

- The value of disk shuffling (reorganizing data layout on disk) is faster access to frequently-used data. The costs include delaying disk accesses while data is being shuffled.
- The value of flushing the contents of a non-volatile write-behind buffer to disk during idle periods is reduced exposure to failures, greater capacity to absorb future write bursts, less interference with foreground IOs, and better disk-arm scheduling for the writes. The disadvantages include potential interference with foreground reads, unexpected head movements, and greater write traffic.
- The value of powering down a disk drive is the energy saved; the costs are that ordinary work may be delayed, extra energy consumed during the "recovery" task of spinning up the drive, and the disk lifetime reduced by repeated start-stop cycles.

Other costs are specific to the idle task: delaying cache flush operations can increase cache space use, decreasing its ability to absorb bursts; disk shuffling requires collecting access pattern information during regular operation.

The utility of using idle time depends in part on how well the detector can find idle periods. For example, a detector that does not find much idle time will not save much energy in powering down a disk; a detector that is overeager in declaring idle periods will cause interference between background and foreground work. Thus the *internal measures* of detector performance can help explain why one detector works better than another for some systems.

2.4 Characterizing idle tasks

There are many different uses for idle time, but they mostly fall into four different categories:

- *Required work that can be delayed,* such as delayed cache writes, migrating objects in a storage hierarchy, rebuilding a RAID array after a failure, cleaning a log-structured file system.
- *Work that will probably be requested later,* such as disk readahead, eager function evaluation, collapsing chains of forwarding addresses for mobile objects, eager `make` [Bubenik89].
- *Work that is not necessary, but will improve system behavior,* such as rearranging data layout on disk, shutting down parts of a system to conserve power, checking system integrity, compressing unused data.
- *Shifting work from a busy to an idle resource,* such as choosing the least-loaded network path, compressing data to reduce disk or network traffic.

Idle tasks can also be characterized by how they react to being stopped and started (we call these *granularity* properties):

- *Interruptability:* some idle tasks can be interrupted at any time, and will stop immediately. Others must complete a fixed granule of work before they can relinquish the system resources they are consuming. (For example, a disk write operation must run to completion, while a powered-down device can be restarted at any time.)
- *Work loss:* when some idle tasks are interrupted, they will lose or must discard some of the work that they have performed. (For example a log-structured file system cleaner may have to abandon work on the current segment.) This cleanup process itself may need resources, and some idle tasks have to be followed by a recovery task to put the system back to a consistent state.
- *Resource use:* most idle tasks block foreground work from making progress to some degree. In the extreme, they may completely deny foreground-work access to a resource (e.g., a disk that has been spun down); in other cases, foreground activity simply slows down while the idle task is executing.

Each of these properties affects applications in different ways. For example, high degrees of multiprogramming will probably make a workload more resilient to an idle task that blocks access to a single resource, since there is probably something else useful that can be done while the idle task has the resource.

2.5 Idle task examples

There are many possible uses for idle time. Storage, compilation, user interfaces, and distributed systems all exhibit highly variable workloads—a clue that a system could benefit from idle-time processing. We surveyed a number of them in an earlier paper [Golding95]. Here we present three examples that we will use throughout this paper, together with how our taxonomy classifies them.

2.5.1 Disk power-down

Several people have investigated powering down disk drives on portable computers to conserve energy (e.g., [Cáceres93, Douglis95, Greenawalt94, Marsh93, Wilkes92b]). Using the taxonomy we have developed, the "idle task" is keeping the disk powered off, with the goal of decreasing energy consumption. The initial step is to spin the disk down, and the recovery step is to spin it back up (Table 1).

Table 1: characteristics of disk power-down; numbers are for an HP Kittyhawk disk.

Initial idle task	spin down disk
Idle task	(do nothing): saves 1.5–1.7 W
Recovery idle task	spin up disk: takes 1.5 s @ 2.2 W
Granularity	unit: can be aborted at any time loss: energy to spin up disk (3.3 J) resource: excludes any other disk accesses
External measures	energy saved delay caused to IO operations

For example, during normal operation, an HP Kittyhawk disk [HPKittyhawk92] consumes 1.5–1.7W. When it is spun down, it enters a "sleep" mode that consumes very little energy. When a disk IO request arrives, the disk must be powered up, which uses 2.2 W for 1.5 s (i.e., 3.3 J). Energy consumption will decrease if the savings from the powered-down mode outweigh the energy cost of spinning it up again. For this disk, it can be achieved if the disk is spun down for as little as 2.2 s; larger disks take somewhat longer to recoup the spin-up cost. However, spinning the disk down too often will increase the latency of disk requests and increase the chance of disk failure. A good idle-time policy will balance these conflicts.

Douglis et al. [1995] found that, using simple adaptive timer-based idle detectors, systems could save as much as 50% of the power that would be required to keep a disk running continuously, and that there were smooth tradeoffs between the amount of energy saved and the delay added to user IO operations.

2.5.2 Delayed writeback

The read-write bandwidth of a disk is a scarce resource during a burst of requests. As write buffers increase in size, synchronous read accesses will come to dominate performance in realistic systems because the amount of memory needed to absorb peak write rates is (much) smaller than the quantity needed to cache all reads [Ruemmler93, Bosch94]. The delays seen by reads can be reduced by delaying writes until idle periods, possibly with the help of non-volatile memory [Baker92b, Carson92a, Chen96b, Ganger94b].

Delayed writeback is an example of delayed work. When a write operation arrives, it is saved in the cache rather than written to disk. This consumes buffer-cache space, which may reduce the read hit rate in the cache, or require more memory. When the system is idle, these accumulated writes are flushed to disk in groups of N IO requests. The larger the value of N, the better the requests can be scheduled at the disk [Seltzer90b, Jacobson91]. This flush can potentially delay foreground reads that arrive during the flush. In practice, reads should be given priority over writes [Carson92a]; however, we'll explore the effect of scheduling the reads and writes with identical priority.

This idle task requires no special initialization or recovery actions. A good delayed writeback system minimizes read latency and cache utilization. Table 2 summarizes these characteristics.

Table 2: characteristics of delayed writeback

Initial idle task	(none)
Idle task	flush dirty disk blocks
Recovery idle task	(none)
Granularity	unit: fixed (*N* sectors) loss: none resources: ties up disk
External measures	max cache space needed change in read latency

2.5.3 Eager LFS segment cleaning

In a segmented log-structured file system, blocks are appended to the end of a log as they are written [Rosenblum92, Carson92a, Seltzer93]. The disk is divided up into a number of *segments*, which are linked together to form the log. As blocks are re-written and their new values appended to the log, earlier copies become garbage that must be reclaimed. A *cleaner* task selects a segment that contains some garbage blocks and copies the remaining valid blocks out of the segment. The segment is then marked as being empty so it can be written over later.

The cleaning operation causes a significant amount of disk traffic, and consumes operating system buffer space [Seltzer93]. The disruptiveness can be minimized if cleaning is performed when there is little ordinary disk traffic. However, segments must be cleaned promptly enough that the system does not run out of clean segments, which would force a segment-clean in the foreground.

The cleaning task is characterized by the delay it imposes on ordinary traffic and how often the system runs out of clean segments. Minimizing the delay is best done with an interruptible cleaner that can discard partially-completed operations. Table 3 summarizes these characteristics.

Table 3: characteristics of LFS segment cleaning

Initial idle task	(none)
Idle task	clean one segment
Recovery idle task	discard partially-cleaned segment
Granularity	unit: fixed (1 segment) loss: up to 1 segment, resource: ties up disk
External measures	foreground cleaning time change in read latency

3 An idleness detection architecture

Having presented an overall framework for idle-time processing, we now turn our attention to how to build idleness detectors. We have found it useful to decompose the solution into a number of separate components, each implementing just one part of the detection algorithm. By combining the parts in different ways we can build detectors on a mix-and-match basis to explore a much wider range of design alternatives than would otherwise be the case.

Figure 6 shows the overall scheme. An idleness detector is composed of a number of predictors and skeptics, along with an actuator:

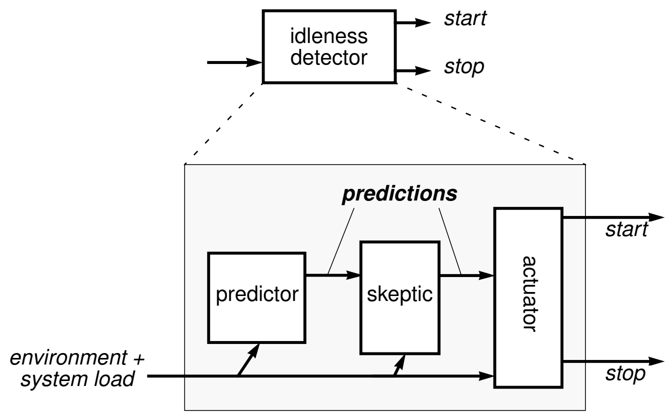

Figure 6: construction of a simple idleness detector.

- A *predictor* monitors its environment—the arrival process, resource usage, and possibly other variables such as the time of day—and issues a stream of predictions. Each prediction is a tuple <*start time, duration*>.
- A *skeptic* [Rodeheffer91] filters and smooths these predictions, possibly combining the results from several predictors.
- The *actuator* obeys the sequence of predictions it gets as input to start and stop the idle task; its purpose is to isolate the interactions with the idle task from the predictors.

The predictors and skeptics form a directed acyclic graph, which we collectively call the *idleness detection network*. Predictors are the leaves, generating streams of predictions for skeptics, which form the internal nodes. The skeptics generate streams of predictions that can be processed by other skeptics, and so on until they read a single node that emits predictions to the actuator.

3.1 Predictors

Since a prediction consists of both a start time and duration, we split the predictor component into two subcomponents. The *start detector policy* generates the start time part of a prediction, and communicates with a *duration predictor policy* to get an estimated duration (Figure 6).

3.1.1 Start time detector policy

Simple detectors use little information to determine when idle periods will start; more sophisticated ones take advantage of knowledge about the arrival process to make better decisions. We present them here in order of increasing complexity.

- *Timer-based:* whenever the system runs out of work, the detector policy begins a timer. If no work comes in before the timer expires, the detector declares that an idle period has begun. The timer period can be *fixed*, *variable*, or *adaptive*. A fixed period does not change. A variable period is computed as a function of some values in the environment, such as time of day. An adaptive timeout period is increased if predictions are too eager, and decreased if they are too conservative.

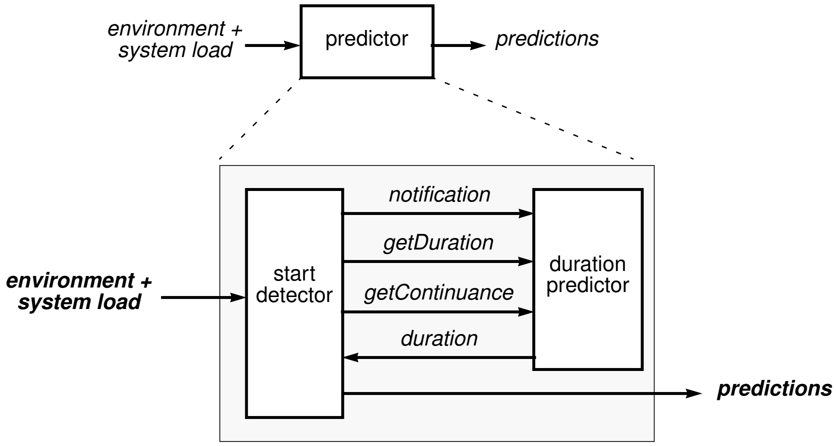

Figure 7: components of the predictor component. The details of the interface are discussed in Section 3.1.3.

- *Rate-based:* the detector policy maintains an estimate of the rate at which work is arriving, and declares an idle period when its rate estimate falls below a threshold. Different threshold rates can be used for "start of idle period" and "end of idle period" to provide some hysteresis. Methods for maintaining the estimate include:
  - *moving average:* the rate is periodically sampled, and the detector computes a moving average of the samples.
  - *event window:* the detector maintains the times of the last *n* arrivals, and estimates the rate as *n* divided by the age of the oldest arrival. This is similar to leaky bucket rate-control schemes for high-speed networks [Cruz92].
  - *time window:* the predictor maintains a list of arrival times more recent than *t* seconds, and estimates the rate as the length of the list divided by *t*. This is a variation on the event window method.
  - *adaptive:* like the other rate-based policies, but the threshold rates are adapted based on the accuracy of recent predictions in order to meet an accuracy goal.
- *Rate-change-based:* these predictors maintain an estimate of the first derivative of the arrival rate to predict in advance when the arrival rate will fall below a threshold.
- *Periodic:* if the workload contains work that repeats with a nearly constant period, a digital phase locked loop or DPLL [Lindsey81, Massalin89a] can be used to synchronize predictions to these periodic events in the workload. By knowing when work will arrive, such as a file-system daemon that does periodic buffer-cache flushes, the idle periods can also be predicted.

3.1.2 Duration prediction policy

A wide range of techniques can be used to adapt an estimate of how long an idle period will last to a changing workload. We list them here according to the amount of information they use about the arrival process.

- *No duration:* no prediction is made (alternately, the prediction is "forever"). Variants on this approach include policies that merely detect the end of an idle period when it happens, rather than making a prediction beforehand. These are most useful when the definition of "idle" allows some residual foreground work.

Table 4: start time detector policies implemented.

Name	Description	Parameters
Timer	Fixed-duration timer	Timeout duration (seconds)
AdaptTimerArithArith	Adaptive timer; increases or decreases duration arithmetically	Duration increment (seconds)
AdaptTimerArithGeom	Adaptive timer; increases duration arithmetically; decreases duration by half	Duration increment (seconds)
AdaptTimerGeomArith	Adaptive timer; increases duration by doubling; decreases duration arithmetically	Duration increment (seconds)
AdaptTimerGeomGeom	Adaptive timer; changes duration by doubling or halving	none
MovingAverage	Rate-based predictor that maintains a moving average of IO/second rate	Threshold rate (IO/s)
PLL	Phase-locked loop that attempts to lock on to 30s Unix sync daemon activity	none
EventWindow	Rate-based predictor that maintains a window of the last *n* operations to estimate a rate	Window size Rate threshold and kind (IO/s, KB/s, or fraction of time busy)

Table 5: duration predictor policies implemented.

Name	Description	Parameters
Fixed	Fixed duration	Duration (seconds)
BackoffArithArith	Increases or decreases estimate arithmetically	Duration increment (seconds)
BackoffArithGeom	Increases estimate arithmetically; decreases estimate by half	Duration increment (seconds)
BackoffGeomArith	Increases estimate by doubling; decreases estimate arithmetically	Duration increment (seconds)
BackoffGeomGeom	Changes estimate by doubling or halving	none
Average	Estimate is a moving average of the duration of recent idle periods	none

- *Fixed duration:* a fixed duration is predicted. The simplest form of this is “enough time to run the idle task once”.
- *Moving average:* the duration prediction policy keeps a moving (possibly weighted) average of the actual durations. The usual average is the mean, but a geometric average or median can also be used. (More generally, the predictor can treat idle periods as an ARIMA process [Box94].)
- *Filtered moving average:* like moving average, but only idle periods greater than some lower-bound are considered during the averaging process, so that the presence of many very short idle periods that cannot be used does not bias the results for longer, useful periods.
- *Backoff:* after each prediction is used, the duration predictor uses the feedback to determine whether the actual duration was longer or shorter than predicted. If it was longer, the next prediction is increased; if it was shorter, the next prediction is decreased. The increases can

be arithmetic, increasing by a constant each time, or geometric, increasing by a constant factor. The skeptic in Autonet [Rodeheffer91] and round-trip timers in TCP [Postel80a, Comer91, Karn91] used geometric increase and arithmetic decrease to maintain a prediction slightly longer than the actual, while a duration predictor works to keep its predictions slightly shorter.

The backoff algorithm can be applied either at the end of the prediction period or at the end of the idle period. The first gives a chance for the algorithm to be much more aggressive in extending its estimates; the latter provides more information, but potentially causes the period to be adjusted much less often.

- *Filtered backoff:* backoff policies that only consider actual idle periods longer than a given lower-bound during their backoff calculations.
- *Autocorrelation:* the autocorrelation on the work arrival process gives the probability of an event arriving or the rate of arrival as a function of time into the future. The predicted duration is the period during which the probability of arrival is below some threshold. The autocorrelation is somewhat expensive to compute, so it might be recomputed periodically rather than continuously. It might also be used to predict the beginning of multiple idle periods.
- *Conditional autocorrelation:* like a simple autocorrelation, except that multiple autocorrelation functions are computed based on some property of arriving events. For example, the expected future might be different following read requests or write requests.
- *Ad-hoc rules:* finally, as with predicting the beginning of an idle period, many systems can take advantage of other specific features of the arrival process, such as periodicity.

3.1.3 Interface between start time and duration prediction policies

In our evaluations, we separated the implementation of the start detector policy from that of duration prediction policy, as shown in Figure 6.

The interface between the two policies required a few revisions to get right. The final version consisted of three parts:

1. Notifications of the beginning and ending of actual idle periods sent from the start detector to the duration predictor.
2. Requests from the start detector for a duration prediction at the *beginning* of an idle period. We called this operation getDuration.
3. Requests for a duration prediction in the *middle* of an idle period, called getContinuance.

Predicting duration of an idle period that is in progress is necessary because many periods run much longer than initially predicted, and it can be useful to get a prediction of how much longer it will likely continue, given that it has already lasted at least as long as the original prediction.

3.1.4 Offline predictors

As with so many problems of this type, optimal idleness detection requires off-line analysis that has knowledge of future events. While this approach is often not useful for building a system, in those cases where usage patterns are stable a one-time analysis may provide a useful prediction. For example, "weekends from 1–6 a.m." is a common time to perform system maintenance.

In practice, however, we have concentrated on on-line detectors for our work.

3.2 Skeptics

A skeptic takes in one or more prediction streams, and emits a new one. Skeptics are used to filter out bad predictions and to combine the results from several predictors into a single prediction stream.

Single-stream (filtering) skeptics include:

- *low-pass:* discards predicted periods that are shorter than some threshold (e.g., the duration of the idle task).
- *quality:* discards predictions from a predictor that is consistently wrong. The skeptic can compute a measure of the predictor's accuracy, perhaps filtered to remove short-term variations, and pass along predictions when the accuracy is above some threshold.
- *environmental:* discards or modifies predictions according to some external event (such as time of day). This can allow idleness predictions to be restricted to times when nobody is around, for example. The time-of-day input can be derived from moving averages of workloads over long periods of time, so this skeptic can be made adaptive.

Perhaps the most important use for skeptics is to combine several prediction streams. For example, a periodic-work detector will not handle non-periodic bursts, while another predictor might. A skeptic could combine the two, only reporting a prediction when both agree.

More generally, a skeptic can combine a number of input streams by weighted voting. Each stream is given a weight, and the skeptic produces a prediction only when the combined weights are greater than some threshold. When the weights are equal and fixed, this becomes simple voting. Alternately, the weights can be varied according to the accuracy of each predictor. This approach has been shown to yield near-optimal prediction in many cases [CesaBianchi94].

3.3 Actuator

The actuator uses the stream of predictions provided by the network of predictors and skeptics to signal idle tasks to start and stop. When the actuator signals an idle task to start running, it can pass along an indication of how long the prediction network expects the system to stay idle. Some of the idle tasks we evaluated used the prediction to scale the amount of work they tried to do.

There is an interesting policy choice related to the actuator: when to signal a task to stop. While the actuator only starts an idle task when the system is predicted to be idle, it can base its decision to stop the task on either the predicted duration, on direct measurements of how busy the system is, or both. The options include:

- Stop the idle task at the end of the predicted duration, ignoring how busy the system becomes.
- Ignore the predicted duration and stop the task when the system becomes busy. If the detector network is accurate, then stopping when the system becomes busy should be almost identical to following predicted duration.
- Stop the idle task at the earlier of either the end of the predicted duration or when the system becomes busy. This approach is the most conservative policy: it ensures that the idle task is stopped before new work arrives when predictions are good, and minimizes interference when predictions are too long.
- Stop the idle task at the later of the end of the predicted duration or when the system becomes busy. This approach guarantees that the idle task will always run for at least as

long as it was told in the prediction, and longer if possible. It is, however, the least conservative, and we have not yet found a use for it.

4 Experimental results

To get quantitative measures of the effectiveness of idle-time processing, we used the taxonomy presented in Section 3 to design and implement a large number of possible idleness detection network components and networks composed from them, whose performance we then evaluated.

We used three idle tasks, as detailed in Section 2.5. They were:

- disk power-down: spin down an idle disk drive to save power;
- delayed writeback: delay disk write operations to idle periods; and
- eager segment cleaning: perform LFS segment cleaning when there is little other traffic.

We begin this section with a discussion of the methods we used in the evaluation, and then study each of the three idle tasks in turn.

4.1 Methodology

We implemented our idle processing architecture in the Pantheon simulation system [Wilkes95]. In particular, we simulated a host system issuing read and write requests to a set of disks. We used calibrated disk models [Ruemmler94], and exercised our detectors using week-long IO access traces taken from three real systems [Ruemmler93] to avoid making simplifying assumptions about access rates or patterns:

- hplajw: a trace of a single-user HP 9000/845 workstation with two 300MB disks.
- snake: a trace of a HP 9000/720 file server at UC Berkeley with three disks—one 500MB and two 1.35GB.
- cello: a trace of the eight disks on an HP 9000/877 cluster server. We often report on cello-disk6, which held the /news partition and accounted for about half the IO traffic in the system.

The Pantheon simulator uses the system model shown in Figure 8. We used an open queuing model, where IO events were replayed according to the times recorded in the trace. To provide support for idleness detection, we modified the DeviceDriver and Disk classes to let us connect idleness detector networks. The delayed writeback and segment cleaning tasks used events from the device driver level, while the powerdown task used events from the disk controller.

4.1.1 Measures used for evaluation

We used two analytical techniques in our evaluation that bear discussion. The first summarizes the tradeoff between two otherwise incomparable measures. The second measures the consistency of a measure across multiple workloads.

All three of the idle tasks we modeled present tradeoffs among multiple measures. The segment cleaning task, for example, balanced aggressive cleaning to keep the amount of unprocessed data low against conservative cleaning to minimize the interference with foreground operations. The ideal was to minimize both the amount of unprocessed data and the interference: a policy that yielded moderately good performance at both measures is better than a policy that yields low unprocessed data at the cost of very high interference.

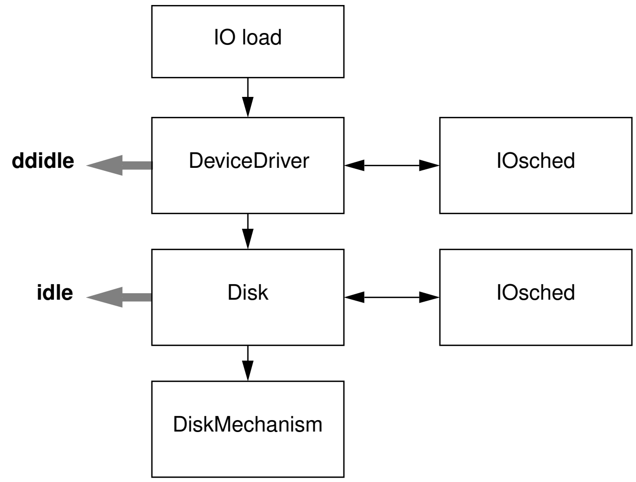

Figure 8: standard Pantheon host-disk system framework, showing the added idle processing attachment points.

We found that a simple Euclidean distance metric summarized this tradeoff nicely. To compute this metric, we first scaled all measurements into a [0,1] range, with 0 being "better" than 1, then took the distance of the resulting point from the origin. The scaling helped to make different measures comparable—one might be comparing megabytes against milliseconds, for example. It also insulated the measure from the peculiarities of different workloads. While it is possible to bias this computation in favor of one measure or the other by adding a scaling factor, we chose to weight the measures equally.

Specifically, consider a sample *(x,y)*, with $x \in X$ and $y \in Y$. Assume that the observed values for X ranged from x_{min} to x_{max} and that larger values of X were to be preferred, so the "best" value for measure X was x_{max}. The scaled value of x is:

$$x' = \frac{(x_{max} - x)}{(x_{max} - x_{min})}$$

which maps all the values in X into a [0,1] range with better values closer to zero. A similar process applies to y. We then computed the distance from (x', y') to the origin:

$$d = \sqrt{x'^2 + y'^2}$$

This formulae work equally well if the best point is the minimum, in which case x_{max} and x_{min} are exchanged in the formulae.

As an example, consider the tradeoffs for the segment cleaning task. The left-hand graph in Figure 11 show the ranges for X (latency improvement, from 16.65 ms to 18.06 ms) and Y (mean queue length, from 0.14 KB to 8455 KB) for the hplajw-disk0 trace. The best value of X is the largest, 18.06 ms; the best value of Y is the smallest, 0.14 KB. A policy that yielded x=17.4 and y=90, near the center of the graph, would be scaled to $x' = (18.06 - 17.4)/(18.06 - 16.65) = 0.46$ and $y = (0.14 - 90)/(0.14 - 8455) = 0.0106$. The distance metric is

$$d = \sqrt{(0.46)^2 + (0.0106)^2} = 0.46012$$

We also wanted to determine how consistently various policies performed across different workloads. A measure of consistency should indicate whether one policy consistently yielded better results than another policy across several workloads. It should also indicate whether a single policy produced equally good results across workloads. We used these indications to suggest policies that are generally safe choices over a wide range of conditions.

Our approach to defining a consistency metric was to use the mean and variance of how well each policy did across the workloads. We first scaled all the results for a particular workload into a (0,1) range just as we did for the distance metric. The mean of this measure for one detector across multiple workloads indicates how close to the best value the detector was, on average, and the variance of the measure indicates how consistent the results were.

More formally, consider a set of measurements $m_{p,w} \in M$ taken by using a set of policies P with a set of workloads W. For each workload $w \in W$, we can find the best and worst values for each workload w, $m_{\text{best},w}$ and $m_{\text{worst},w}$, and compute range of values $r_w = m_{\text{best},w} - m_{\text{worst},w}$. For each policy $p \in P$, the scaled measure is

$$m'_{p,w} = \frac{(m_{\text{best},w} - m_{p,w})}{r_w}$$

This scaled measure can be thought of as the fraction that policy p produced of the best result for workload w. The mean,

$$\overline{m'_p} = \frac{1}{|W|} \sum_{w \in W} m'_{p,w}$$

gives the overall "goodness", and allows us to compare two policies; the variance of this measure indicates whether the policy performance varied across different workloads or not.

4.2 Disk power-down

The first idle task we looked at was disk power-down, which tries to save energy by turning off a disk drive when it is not being used, as discussed in Section 2.5.1.

There were two external measures we used to evaluate the disk power-down idle task: the energy saved, and the number of operations that had to be delayed because the disk wasn't ready when the operation was issued. These are similar to the measures that have been considered in other studies on this problem [Douglis95].

In our model, the idleness detection network monitored activity inside the disk controller. Each disk was augmented to include an idleness detection network and an idle task. The idle task initiated a spindown whenever it received a prediction from the detection network, and waited to spin the disk back up until a disk IO request arrived.

Real disks have a great many different power-management systems. We used a simple, representative power management system derived from typical 2.5" disks. We assumed that the disk would require 1 s to spin down and 1.5 s to spin up. It would consume 1.6 W during normal operation; 0.4 W while sleeping; and 2.4 W while spinning up. For these parameters, the break-even point, where the energy saved while being spun down equals the energy cost of spinning back up, is 3 s.

For the idle detection networks, we evaluated each of our start time detectors combined with a Fixed 10 s duration predictor—this being what we considered a reasonable minimum duration. The fixed duration predictor provided a target for the adaptive start time detectors: when working properly, they adapted their detection policy to only declare idleness when it was likely that the system would stay idle for at least long enough to save a little power.

Figure 9 shows the detailed results for two of the disk traces we evaluated, comparing the power saved against the number of delayed operations. Note that higher power savings and fewer delayed operations is to be preferred, which is to the lower right in the graph. The small graphs show how each family of start time detection policies performed compared to the overall picture.

There is a clear tradeoff between the two measures. In our implementation, at least one operation was delayed every time the disk was powered up because the disk never tried to anticipate when future requests might arrive. Detectors that powered down the disk more often therefore delayed the most operations, and indeed for all but three disks in the traces the two measures are correlated at a 95% likelihood. The three exceptions came from disks where the PLL detector saved relatively little power but still delayed many operations. (The sample in the lower left of both graphs in Figure 9 is from a PLL detector.)

For power savings, a few start detectors consistently did the best over the 13 disks in our workload traces, as shown in on the left-hand side of Figure 10. The AdaptTimerArithGeom 0.1s and 1.0s and Timer 0.5s and 1.0s appear to be the four safest choices: they get within 2% of the best power savings for all of the workloads. They are our recommended choices when power savings are most important.

The rate-based (EventWindow and MovingAverage) detectors produced better than anticipated results for this idle task. We expected them to do poorly because they were intended to find periods of low traffic, while disk powerdown requires periods of no traffic. In practice, it appears that the EventWindow detectors with small windows and lenient thresholds do fairly well.

The detectors that delayed the fewest operations, as shown on the right-hand side of Figure 10, were the most conservative: rate-based detectors with low rate thresholds and large windows. These detectors delayed between a tenth and a third as many operations as did the detectors that yielded the best power savings.

4.3 Delayed writeback

The next idle task we looked at was delayed writeback. The overall performance of a disk system can be improved by processing read operations immediately, since processes are waiting for their completion, while delaying write operations a bit—in particular, delaying them to times when there is no read traffic occurring. If this is successful, it will the latency of read operations will be smaller, since they will be able to proceed without interference from write traffic. The cost is that the data to be written consumes memory resources while it is waiting. A good idleness detector for delayed writeback will minimize the interference between read and write operations while keeping the amount of unwritten data at a minimum. Thus the two measures we used to evaluate the delayed writeback task were the improvement in read latency and the length of the write queue.

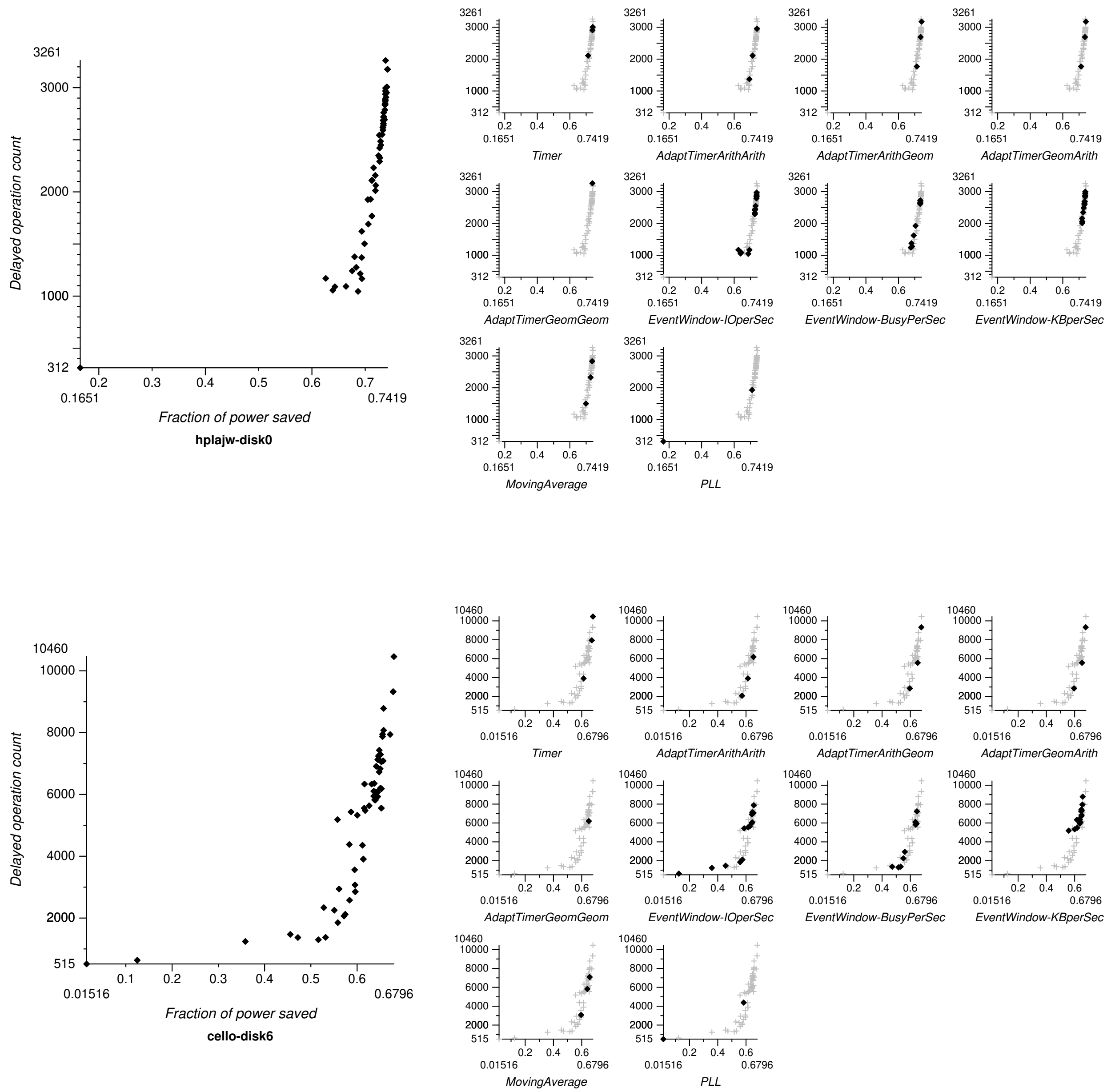

Figure 9: disk powerdown: power savings versus number of delayed operations. Better is lower and to the right.

We implemented this idle task by inserting a special "write delay" device driver between the host workload source and the normal device driver. The write delay device driver sent read operations directly to the normal device driver for immediate service. It placed write operations on a FIFO queue, from which they were removed when the idle task was informed that there was enough time to do some writes. This was a simplistic model of delayed write operations: it did not attempt

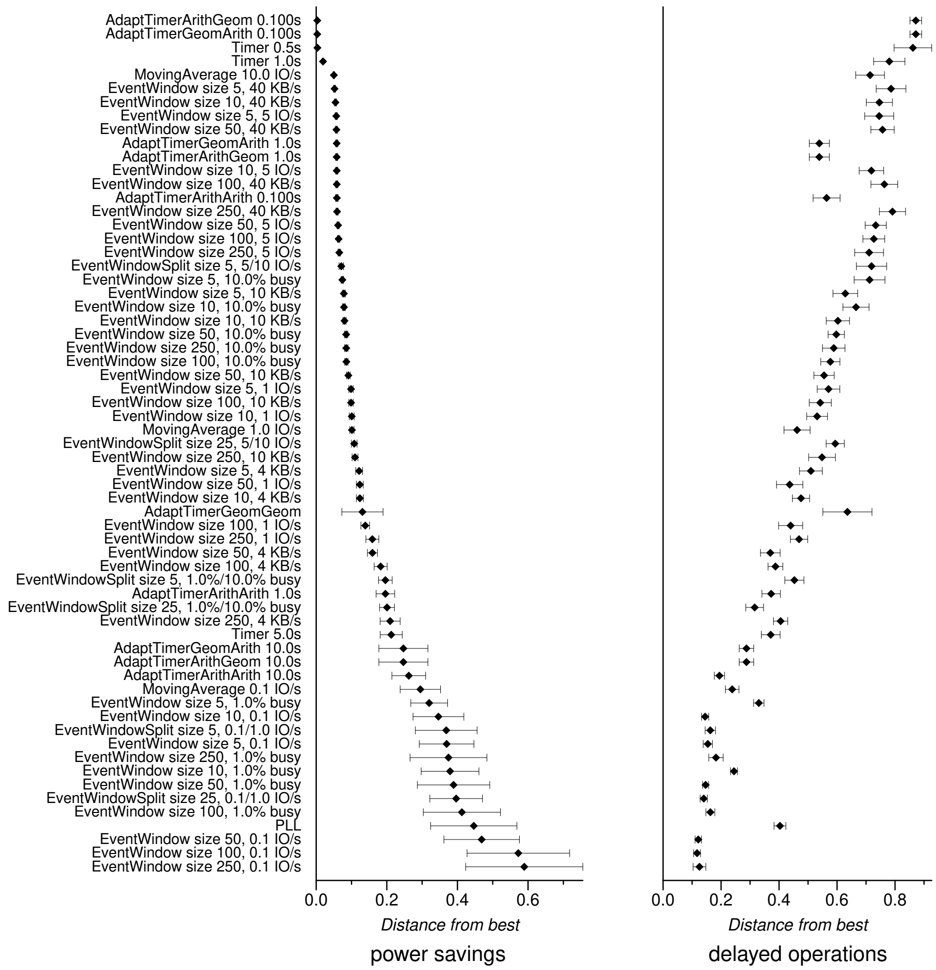

Figure 10: disk powerdown: overall consistency of power savings and operation delay for the detectors.

to coalesce overlapping write operations in an attempt to reduce the amount of data to be written; nor did it attempt to satisfy read operations from the data in the queue. We believe that real performance would be slightly better than indicated here, but as with our other models, this one is sufficient to investigate idleness detection.

The idle task wrote items to disk in the order they were queued when it was informed there was idle time. It would issue only as many write requests as could be processed in the predicted idle duration, and it allowed only up to 8 Kbytes of outstanding write requests in order to limit the recovery time when an idle period ended. The system also enforced a maximum queue size of 16 Mbytes; if more writes were enqueued than that, a foreground task would be triggered to flush at least 8 Mbytes of data from the queue.

Figure 11 shows the relationship between the improvement in read latency and the length of the delayed write queue for the hplajw-disk0 and cello-disk6 workloads.

For the hplajw-disk0 workload, there was plenty of idle time available and so read latency is improved by up to 18.1 ms (39.7%). There was also usually sufficient time after a burst of writes to flush the queue, so the mean queue length was short.

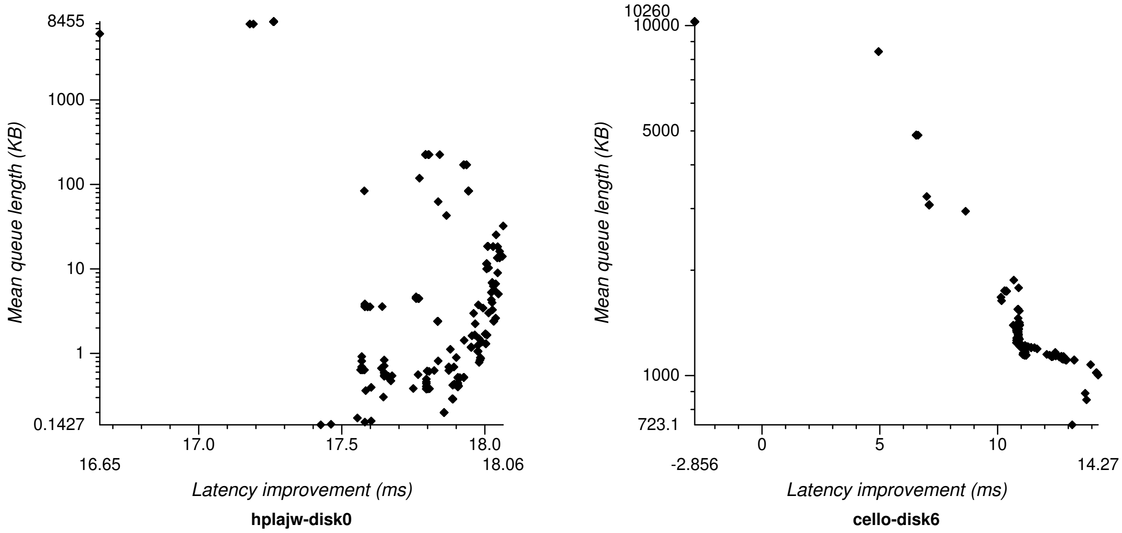

Figure 11: delayed writeback: relation between operation delay and mean delay queue length. Better is to the lower right.

In the cello-disk6 workload, on the other hand, there was much less idle time available. Nonetheless, most detectors provided an improvement in read latency, up to 14.3 ms (18.4%). All of the detectors reached the maximum queue length at least once in the cello-disk6 workload.

For the busy cello-disk6 workload, a group of detectors, all non-adaptive Timers with timeouts of 5 s or longer that used adaptive duration predictors, made read latency worse, by about 4%. For the less busy hplajw-disk0 workload, all detectors improved performance by at least 37%—though the same cluster of non-adaptive Timers showed up among the least effective.

We were curious how consistent the choice of best overall detector was among the workloads we investigated. We computed the scaled Euclidean distance metric defined in Section 4.1.1 for each run to determine how well a particular policy balanced queue size against latency improvement. We then used the mean and variance of the distance metric across different workloads as the measure of the overall usefulness and consistency of the policy. Figures 12 and 13 report the results, grouped by start detection policy and duration prediction policy.

There are a few lessons to be learned from these results. First, the duration prediction policy does not matter nearly as much as the start detection policy. For many start detection policies (Figure 12) the results for all detector-predictor combinations are clustered closely about the mean, while the results for duration predictors (Figure 13) show large variations. Our implementation only used the duration prediction to try to determine how much data to write out at any given time, and we believe that most of the idle periods were longer than needed to empty the delay queue.

The overall best policy combination was not the combination of the generally best detector and the generally best predictor. The overall best combination was the AdaptTimerArithArith 10s + Fixed 100s policy: while the Fixed 100s prediction policy was ranked the best overall predictor, the adaptive timer detector policies were not.

Overall, however, the rate-based (EventWindow and MovingAverage) start detector policies showed less variance than the adaptive timer policies. Our conclusion is that a rate-based policy combined with a simple fixed-duration predictor is a good choice for the write delay idle task.

4.4 Eager LFS segment cleaning

Our final idle task was designed to model segment cleaning in a log-structured file system: the system must periodically perform garbage-collection operations to compact all the active data together [Rosenblum92]. This workload introduces extra housekeeping work into the disk subsystem, beyond the work required for ordinary foreground reads and writes. The goal of using idle time is to perform these housekeeping operations at times when there are few foreground operations. However, the housekeeping has to be performed often enough to avoid running out of free space into which new data can be written.

We did not model segment cleaning per se, since we were using traces recorded from systems that did not use a log-structured file system. Instead, we constructed an additional workload that periodically introduced a burst of read and write traffic similar to a cleaner reading a segment into

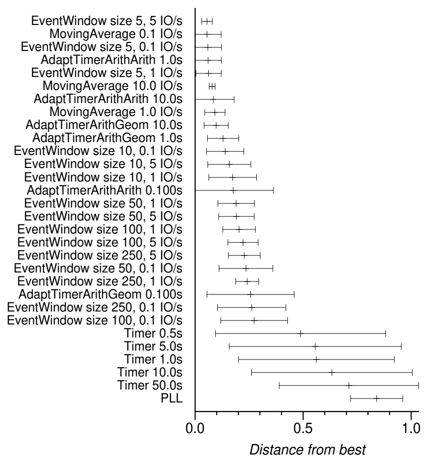

Figure 12: delayed writeback: comparative performance of various start detection policies. The range bar shows the mean distance from the best observed result and standard deviation of that distance across all workloads for all runs, with all the duration predictors in Figure 13, that used the named detection policy.

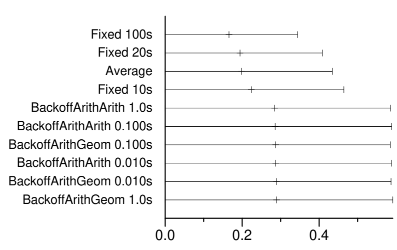

Figure 13: delayed writeback: comparative performance of various duration predictor policies. The range bar shows the mean distance from the best observed result and standard deviation of that distance across all workloads for all runs, with all the start detectors in Figure 12, that used the named detection policy.

memory, then writing it out elsewhere. We modeled the amount of data that needed to be copied by assuming that the file system was in a steady state, so that each byte written created one byte of data that must be copied. While this is not strictly what an LFS cleaner does, we believe that the resulting workload was sufficiently like an LFS cleaner to evaluate idle detector performance.

Specifically, the cleaner task repeatedly executed a *cleaning cycle* whenever the system was idle. In one cleaning cycle, the task read up to 1MB of data consecutively from one location on the disk, then wrote the data to another location on the disk and advanced the read and write locations for the next cycle. The cleaner task used the duration prediction from the idleness detection network to compute how much data could be processed in one cycle without interfering with foreground work, and set the amount read in the cycle accordingly. The task would only begin a cycle if it expected that it could process at least 64KB without interruption. If the cleaner was told by the actuator to stop, it immediately wrote out as much as it had read and then stopped. Interruptions during the write portion of the cycle were ignored.

The interference between cleaning and foreground operations was measured by the delay imposed on the foreground operations. This should, of course, be minimized. Most studies on LFS segment cleaning policies have compared the overall service times using different policies—for example, comparing background cleaning with cleaning on demand in the foreground [Seltzer93]. For this study, however, we were not modeling an entire log-structured file system and instead measured interference by the difference in mean service time between the system without cleaning and background cleaning with various idleness detection policies.

Whether housekeeping is being performed often enough was measured by the amount of unprocessed data. Since we were not modeling garbage collection per se, we instead measured the amount of unprocessed data. The lower this amount, the more space was ready to absorb a burst of writes, and hence the less likely it was that a real LFS would have to perform garbage collection in the foreground.

We report results from the hplajw and cello traces. Since hplajw was lightly used, the amount of unprocessed data stayed low and the cleaner task had ample opportunity to run. The cello system was much busier, so the cleaner had both more work to do and the idle periods were shorter. We varied both the start detection and the duration prediction algorithms since this idle task was sensitive to duration predictions.

For both workloads, most idle detectors worked well enough to cause a negligible increase in mean operation delay. For cello-disk6—the busier and thus more difficult system—the worst detector delayed user operations by an average of 5.2 ms, which was only 1.4% of the original mean operation service time. (In this trace, many write operations were part of large bursts and the operation service time included the time spent waiting for preceding operations to complete.) For hplajw-disk0, a lightly used disk, the delay due to the worst detector was slightly less good: 6.6 ms, or 6.2% of the original service time.

For the busy cello-disk6 workload, the mean amount of unprocessed data ranged from 3.6 MB to 84 MB; a few detectors never declared idleness and thus cause unbounded accumulation. A real LFS would have had to perform foreground cleaning in those cases. For the hplajw-disk0 workload, unprocessed data ranged from 0.42 MB to 2.0 MB, with about half the detectors yielding a mean of less than 0.5 MB. The start detectors that never found idle time were all fixed timers. Some had too long a timeout value; others used an adaptive duration predictor that did not receive enough accurate detections to provide usefully-long duration predictions.

Figure 14 shows the relationship between operation delay and mean amount of unprocessed data for two of the disks we evaluated. While one might expect a tradeoff between operation delay and unprocessed data amount, some detectors do well at both measures.

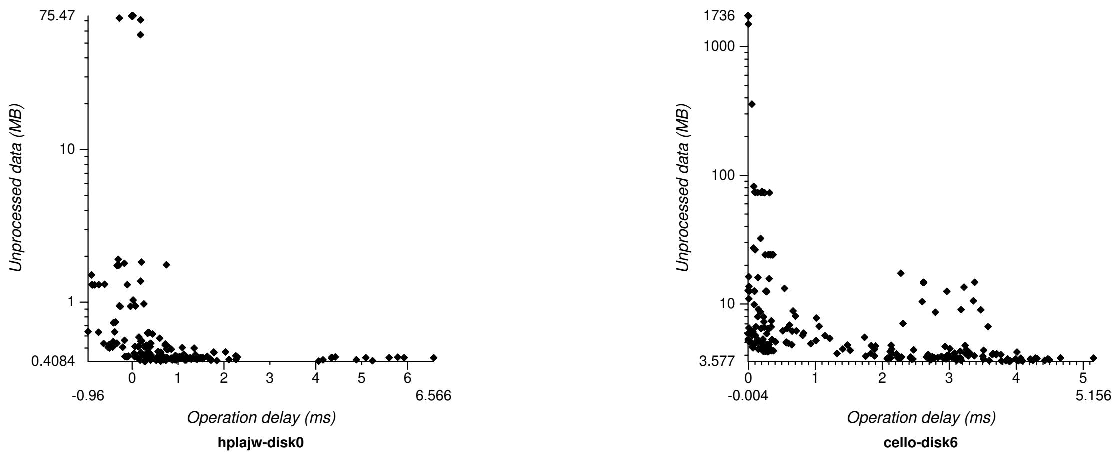

Figure 14: relationship between operation delay and mean amount of unprocessed data. Some policies actually improved the mean operation delay because of the difference in disk head position from the added work.

Figure 15 shows how the different start detection policies performed over the various workloads. The rate-based start detectors were generally the best. We only used measures from those runs that performed cleaning to determine the scaling range, and so the outlier policies show in the graph as having mean distances much greater than one. We also evaluated the ranking when ignoring the outlier results; rate-based detectors were still generally the best.

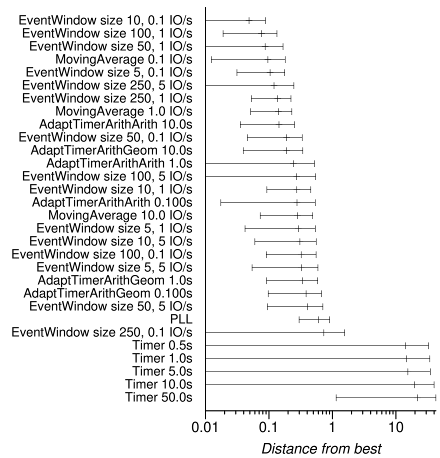

Figure 15: stability of the various start detector policies. The range bar shows the mean and standard deviation of all policies that used the detector. Recall that outliers were removed before scaling, so policies that never performed cleaning resulted distances greater than one.

Rate-based detectors work well for this idle task because allowing a few foreground operations during a cleaning cycle allows the system to find more time for cleaning than if a stricter policy were used—such as a Timer-based policy, which requires absolute idleness in the system. At the same time, by requiring the IO rate to drop below a fairly low threshold (0.1 or 1 IO/s), the policy ensures that few foreground operations will be affected.

We expected that duration predictors would be important for the cleaning task, since it limits the amount of work it will attempt in one cleaning cycle so that it can likely be completed within the predicted idle duration. However, as Figure 16 shows, there was no statistically significant difference in results between duration predictor policies with the possible exception of the Fixed 100s policy.

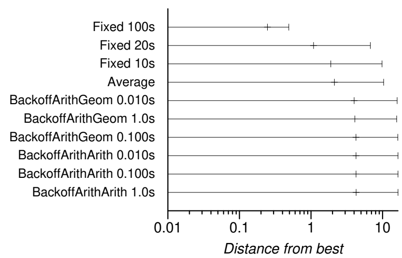

Figure 16: stability of the various duration predictor policies. The range bar shows the mean and standard deviation of all policies that used the detector. There is little statistical difference between policies.

We conclude that the choice of when to start cleaning is more important than the accuracy of predicted duration, and that rate-based detectors do the best job for this kind of idle task.

4.5 Summary

In this section we have shown that idle detection can be used successfully to guide the execution of three different idle tasks: powering down a disk drive, delaying write operations, and performing LFS cleaning operations. The disk powerdown experiments showed a significant power savings (60-70%) over letting a disk continue spinning, and we found that simple, adaptive methods are as good as the best fixed timeout values often used for this task, and sometimes better. The delayed write results showed that idle time can be used to move operations that are not time-critical out of the way of operations that are, resulting in 20–40% improvements in service time for time-critical operations. The cleaning task showed that housekeeping operations can successfully be moved to periods when the system is being lightly used, while not falling behind on them.

We found that the policy that detected when the system is idle had more effect on the performance results than the choice of duration predictor. We had expected that the tasks that use duration predictions to guide the amount of work they try to accomplish in one idle period would be sensitive to the duration predictor, but these tasks appear to complete their work quickly enough that duration predictions do not need to be particularly accurate.

Finally, we found that timer-based detector policies worked well for the task that required complete idleness (disk powerdown), while rate-based detectors worked better for the tasks that could continue while foreground operations executed.

5 Analysis

The taxonomic approach we have taken is different from most previous investigations into using idle time. Most of the policies that the taxonomy describes adapt in some way to system behavior, and most have one or more tuning parameters. In this section we investigate why a taxonomic approach was beneficial, and why adaptivity was important. We show that performance was not very sensitive to tuning parameters, suggesting that making a somewhat wrong tuning choice is not usually catastrophic.

We also investigated how the performance of an idle task using a particular idleness detection policy is related to internal measures of how well that policy can find and predict idle time. The results suggest that external performance measures correlate with internal measures only when either idle time is difficult to find, as in a busy workload, or when the penalties of an incorrect prediction are high. For other cases, the external performance of the idle tasks is usually very good and so the choice of policy does not matter so much.

We have one negative result: we investigated whether skeptics (Section 3.2) could improve performance. The skeptics we investigated did not.

5.1 Effectiveness of the taxonomy

Taking a taxonomic approach to analyzing idleness detection was worth the effort. We saw two costs: the time spent developing the taxonomy, and the computing time spent evaluating the large number of detection policies we generated using the taxonomy. The benefits came from the coverage of the problem space that the taxonomy provided, and the software-engineering benefits of a module design guided by the taxonomy.

The cost involved in determining the components was small—a few days discussion among two people, spread out over half a month. The implementation effort was also low: one person completed the first version of the entire idle processing code except the PLL detectors in about two weeks, including the time spent learning the Pantheon simulator system. The final code evolved over the next few months, overlapped with other work. The final version of the entire system, including the implementation of the three idle tasks, consists of slightly more than 11k lines of C++ code and 600 lines of Tcl code.

As a benefit, the taxonomy adequately covers the kinds of idleness detection we have seen used in practice. We have been pleased to find that all of the work that we are aware of that has been published since our first paper [Golding95] has fallen neatly into this taxonomic structure. For example, Douglis et al. [1995], while developing adaptive disk power-down policies, investigated the use of adaptive timers with either arithmetic or geometric adjustment. The primary differences between their policies and our AdaptTimer policies are that they considered using different increment values for increase and decrease; they limit the smallest and largest timer values; and the decision to adjust the timer depends on an application-specific condition (whether recent spindown activity has met a user goal or not). Likewise, Helmbold et al. [Helmbold96] have investigated using a skeptic based on machine learning techniques to choose the best from among a family of fixed timer detectors. In both these examples, we found that having a taxonomy helps identify the essential characteristics of their work, separate from the specific problem that they were investigating, and suggest ways that the ideas can be generalized or used for other problems.

Dividing the problem into smaller components made implementation easier. The usual software engineering arguments for modularity applied to this problem: the individual start detection, duration prediction, and skeptic objects are quite simple, often consisting of less than twenty lines of non-boilerplate code. Being able to combine different policies at runtime meant we could investigate a large policy space with a small implementation effort.

Using a taxonomic approach helped us to find better policies than we might otherwise have found. Consider how different the ranking of idleness detection policies is among the three sample idle tasks. If we had considered only, say, the adaptive timer detectors—which are best for disk powerdown and delayed writeback—we would not have found any of the overall best detectors for segment cleaning, which are all rate-based. Moreover, the absolute best idleness detection policy combination for delayed writeback across all the workloads did not use the best overall detector start detection policy, which indicates that it was necessary to look at all the start detector/duration predictor combinations.

The taxonomy also generated some surprises. For example, we had previously believed that the EventWindow detectors would work better with long windows, in order to remove the effects of transient burst behavior. The measured results differed from our expectations, and on further investigation we found that a long window did not allow the detector to react quickly enough when the system stopped being idle.

5.2 The importance of adaptation

Many of the idleness detection and prediction policies we have investigated try to adapt to the system they monitor. Such adaptation has two purposes: it reduces the sensitivity of an initial design to later deployments, and it allows a running system to adapt to changing requirements.

To illustrate the importance of adaptation, we performed a simple analysis on our traces. Consider a system where the "benefit" b of using an idle period of duration d is quantifiable as $b = 1.2d - 3$, which is a simplification of the energy savings in Joules from the disk power-down idle task. If a fixed timeout policy of length t is used to detect the start of an idle period, then the overall benefit becomes $b = 1.2(d - t) - 3$. We performed an offline analysis to find the timeout period that maximized this benefit, using traces from four systems: the three systems we used for evaluating idle tasks in Section 4, plus a one-hour trace of a very busy transaction-processing system. We considered both the timeout value that maximized the benefit over each entire trace (the *trace-optimal* timeout), and the optimum values for each hour subset of the traces (the *hourly-optimal* timeout).

Figure 17 shows that there is significant opportunity for adaptation: the hourly-optimal timeout period varied widely over time—on the quiet hplajw-disk0 trace, the optimal timeout ranged from 50 ms to 1 s.

That such opportunity for adaptation matters in practice is shown in Figure 18, which shows how the benefit derived from using an adaptive policy, a Fixed 1s policy, and the trace-optimal fixed timeout policy compares to the hourly-optimal fixed timeout policy. The adaptive timer policy was an AdaptTimerArithArith 0.1 s, which incremented or decremented the timeout duration 0.1 s at a time.

These results support our argument that adaptivity is important for building a system that can be used for many different workloads. The Fixed 1s timer almost always performed suboptimally, and sometimes it was substantially worse than the hourly optimal (for two of the database traces). The adaptive policy, on the other hand, performed much closer to optimal on those same traces.

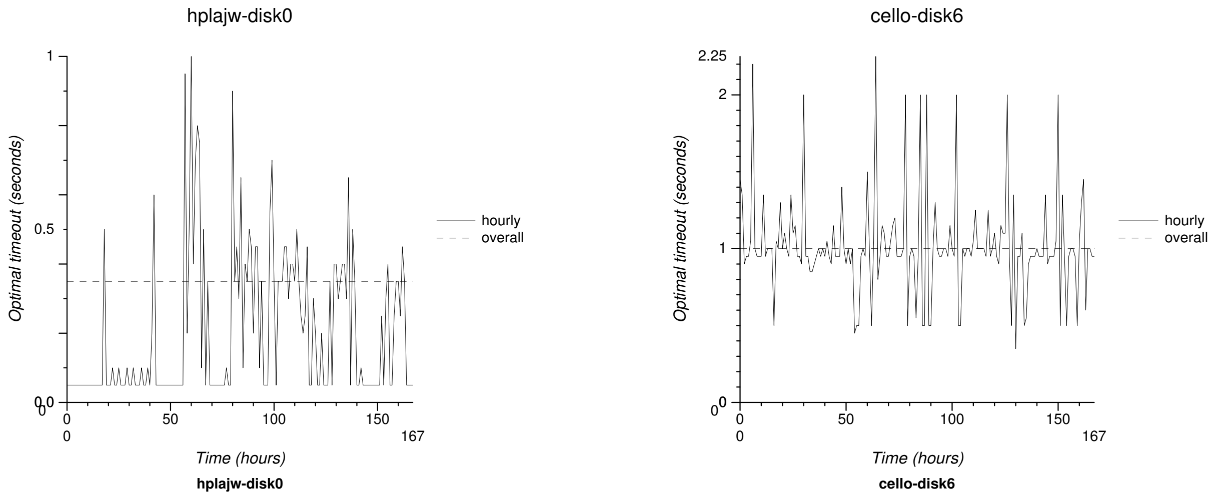

Figure 17: optimal fixed timeout values for two traces.

Adaptivity is also useful in tracking short-term changes within a workload, and an hourly granularity is too coarse. The adaptive policy almost always did as well or better than the hourly-optimal fixed timeout policy, which in turn did better than the trace-optimal fixed timeout.

When we changed the benefit formula so that the break-even point was much longer than in our previous formula, $b = 1.2(d-t) - 15$, the differences become more pronounced, as shown in the lower graph in Figure 18. While the Fixed 1 s timer policy sometimes got equal or higher benefit than the adaptive timer, the adaptive timer was more consistent. In two of the database traces, the best benefit came from never using idle time. These bars are omitted from the graph.

Our conclusion is that adaptivity helps to make idleness detection resilient to differences in workloads and to changing workloads, in many cases yielding results close to optimal.

5.3 Relations between internal and external measures

When building a system, one would like to find a good idleness detector without performing the exhaustive evaluation we did for our three example idle tasks. Ideally, one could make a prediction of which detectors will work well based on their internal measures when measured against a similar workload.

In the following sections we will first define a number of internal measures, then look at how the external measures for our example idle tasks were related to them. The results from our investigation are mixed: we found correlation between internal and external measures for some of our applications, but not for others.

5.3.1 Internal performance measures

Internal measures quantify how well a particular detector finds idle periods, and are independent of a particular idle task.

Taking the internal measurements depends on an application-specified level of acceptable idleness. The simplest definition was that the system should be considered idle whenever there were no requests active. We used this definition because it is trivial to detect on-line. A more

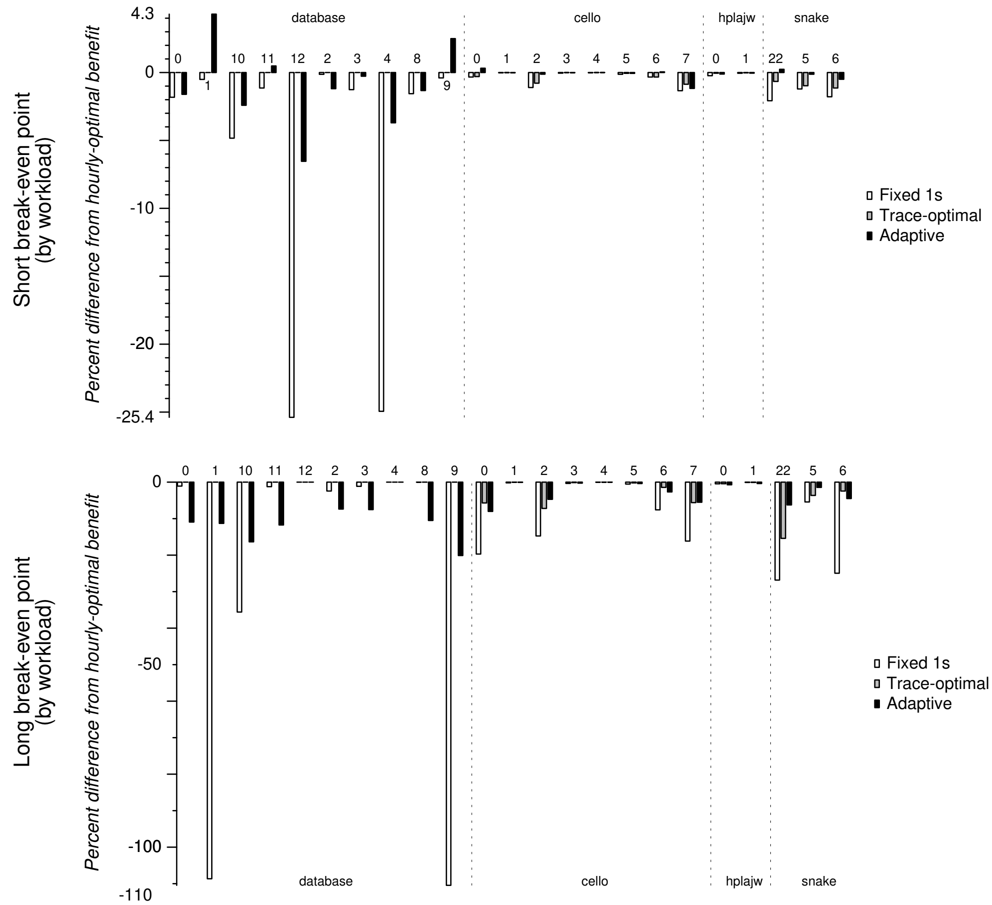

Figure 18: percentage ratio of benefits from using various timer-based start detection policies to that of the hourly-optimal fixed timer policy, for various workloads.

complex system might use low operation rates to define idleness, but such a definition often requires off-line analysis to detect properly.

We start with several primitive measures:

- The *predicted* time is the total amount of time a detector declared to be idle.
- The *actual* time is the total time the best possible off-line detector could produce.
- The *overflow* time measures the amount of time that the detector declared as idle when the system was not idle.
- The *violations* count the number of operations that overlapped the declared idle periods. The *violation rate* is the number of violations per second of declared idle time.

From these basic measures we compute two derived measures:

- The *efficiency* of a detector: this is a measure of how good the detector is at finding idle periods in the workload. It is defined as *efficiency = (predicted - overflow) / actual*

That is, it is the fraction of actual idle time that was declared idle.

- A detector's *incompetence* evaluates how much of the predicted idle time is not idle, penalizing over-eager detectors. It is defined as

incompetence = overflow / actual

A good idleness detector will have a high efficiency and a low incompetence.

5.3.2 Relations

We investigated the relationship between internal and external measures by measuring the efficiency and violation rate produced by each idleness detection policy, then determining whether the results for that policy correlated with the external performance measures for the three different idle tasks.Table 6 summarizes the results. We measured the relationships between measures by looking for correlation at a 95% confidence level.

Table 6: relations between internal and external performance measures.

External measure	Efficiency	Violation rate
Powerdown:		
Energy	yes	yes, except for PLL detectors
Operation delay	yes, except for PLL detectors	yes
Delayed writeback:		
Mean queue length	yes	no
Latency improvement	yes (only busy workloads)	no
Distance metric	yes (only busy workloads)	no
LFS segment cleaning:		
Mean unprocessed data	yes	no
Operation delay	yes	no
Distance metric	yes, except for some Fixed timers	no

- *Disk power-down:* we found close correlations between the two external measures—power savings and the number of delayed operations—and both efficiency and violation rate. There was one exception: the PLL detectors simply did the wrong thing most of the time. Efficiency is correlated with power savings because an efficient detector is finding most of the opportunities to have the disk powered down. Violation rate is correlated with power savings because the length of time spent powered down in any one idle period was determined by the time until the next operation arrives, which is on average the inverse of the violation rate.

- *Delayed writeback:* the violation rate does not appear to be related to any of the external measures we took. Delayed writeback is relatively insensitive to occasional overlapped foreground and background operations. We could not establish correlations with efficiency for the traces from quiet disks, because nearly all idleness detection policies yielded efficiency greater than 90%. For the busy cello-disk6 workload, however, efficiency clearly correlated with all three external measures. This is because an efficient idleness detection policy will find more time to flush the delay queue than an inefficient detection policy.
- *Eager LFS segment cleaning:* once again, the violation rate was not related to the external measures because it was rare for a cleaning cycle to run long enough be interrupted by new work arriving—for most idleness detection policies, less than 5% of cleaning cycles for the busy cello-disk6 workload, and much lower for quiet disks. Efficiency, on the other hand, was correlated with all three external measures, with the exception noted in Section 4.4 of a group of fixed Timer detectors that did especially poorly.

These results suggest that internal measures are correlated with external one only when there is little idle time available or when the penalty of getting a poor prediction is high. Our conclusion is that internal measures may be helpful in finding a good detector for heavily-loaded or pathological situations, but that most of the time the problems are undemanding, the results are uniformly good and hence there is little guidance to be had from the internal measures.

5.4 Sensitivity of internal measures to policy parameters

Most families of idle detection mechanisms in our taxonomy can be tuned to provide different behaviors. We investigated two example policies—one start detector and one duration predictor—to determine how sensitive their internal performance measures are to their tuning parameters. We found that they were generally well behaved: while there usually is an optimum parameter setting, the results from nearby settings were not much worse. We also found that the results, when they matter, are similar between busy and quiet workloads.

5.4.1 Start detection policy parameters

We investigated tuning the EventWindow+IO rate threshold detector because its external performance varied depending on its parameter settings. For example, at low rate thresholds the detector yielded good power savings for disk powerdown, but high rate thresholds did poorly.

The detector is parameterized by rate threshold and window size. The rate threshold governs how quiet the system must be before being declared idle. We expected a low threshold to give low efficiency, because it would take some time for the mean IO rate to drop below the threshold, and the waiting period could have been used profitably. A low threshold should, however, give a low violation rate. The window size governs responsiveness: a small window means the detector reacts quickly to changes, but it may prematurely declare idleness. A large window may cause the detector react too slowly to start of idleness and to the start of a burst of requests.

To measure the effects of tuning a detector, we set up an experiment to take internal measures with a no-op idle task and a Fixed 10 s duration predictor. The actuator was parameterized to follow the predictions exactly: when it received a prediction, the idle task ran for 10 s, whether new work arrived or not. We took measurements from a quiet disk (hplajw-disk0) and a busy one (cello-disk6).

On the quiet disk, the violation rate was relatively insensitive to the window size. Bursts of requests were small and infrequent, so there was plenty of time for the average rate to drop below any threshold we tried. As one might expect, the measured violation rate was closely related to the threshold rate parameter setting.

On a busy disk behavior was more complex, as shown in Figure 19. Medium-size windows (25–250 requests) gave low violation rate results at high rate threshold settings. This is probably because these window lengths balanced between responsiveness at the start of a burst—optimized by a short window—and conservatism in declaring idleness. For rate thresholds below about 0.1 IO per second, the smaller windows performed better. It took a long time for the average rate to fall below the threshold, no matter what the window length, so the system was already conservative; using a small window improved the responsiveness when a new burst started.

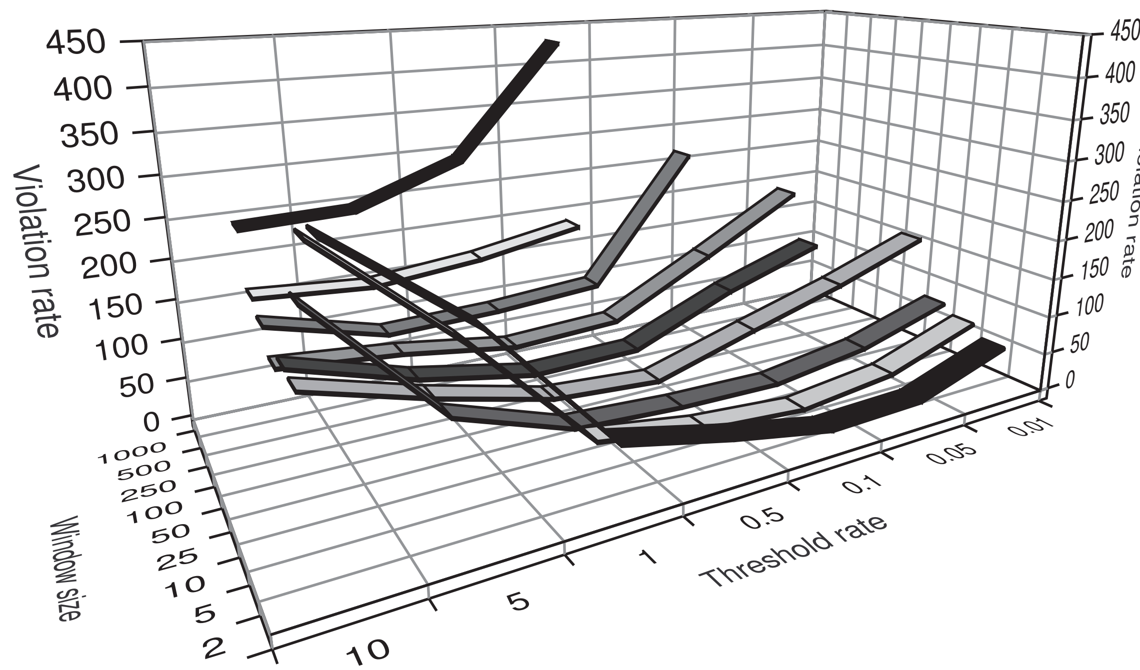

Figure 19: violation rate for EventWindow detector on trace cello-disk6, varying window size and threshold rate.

For a given window size, the lowest violation rate came with a threshold related to window size, as shown in Table 7.

Table 7: the threshold rate that produced the lowest violation rate for various window sizes on the cello-disk6 workload.

window size (requests)	2	5	10	25	50	100	250	500	1000
best rate threshold (IO/s)	0.1	0.1	0.5	0.5	1.0	1.0	5.0	5.0	10.0

Other internal measures, such as efficiency, confirmed our expectations. Lower rate thresholds yielded lower efficiency, since the detector was more conservative about declaring time idle. On busy disks, efficiency decreased with larger windows, since the detector was slower to respond to the end of a burst.

5.4.2 Duration predictor policy parameters

The second tuning experiment investigated how sensitive internal measures were to changes in the duration increment parameter of the BackoffArithGeom predictor. This predictor works by maintaining a current estimate of the next duration. When an actual duration is longer than the prediction, the predictor increases the next prediction arithmetically by adding the increment value. When the actual duration is shorter than the prediction, the predictor cuts the next prediction in half. The intent was that this predictor would be conservative: it would slowly increase its predictions but quickly decrease them to avoid interference with other work.

In the experiment, we investigated increment values ranging from 0.5 ms to 80 s, and measured the efficiency, violation rate, and incompetence observed by strictly following the predictions. We used the AdaptTimerArithArith 0.1 s start detector since it performed well for some external measures (power savings for disk powerdown, mean unprocessed data for segment cleaning) and generally gave high efficiency. This detector adjusts its timeout value fairly slowly, so the start detector and duration predictor should not interfere too much with each other's adaptations.

Figure 20 shows the relationship between the duration increment value and efficiency a typical disk. The best efficiency generally came with small increments, at or below 2 or 3 seconds. The efficiency was very good up to a threshold point, then dropped significantly. Any value in the range of 0.05 to 1 second appears to produce about equal efficiency. With larger increments, we believe the duration predictor was too quick to increase the prediction, and that the system entered a cycle of expanding the prediction in one period then contracting it drastically in the next.

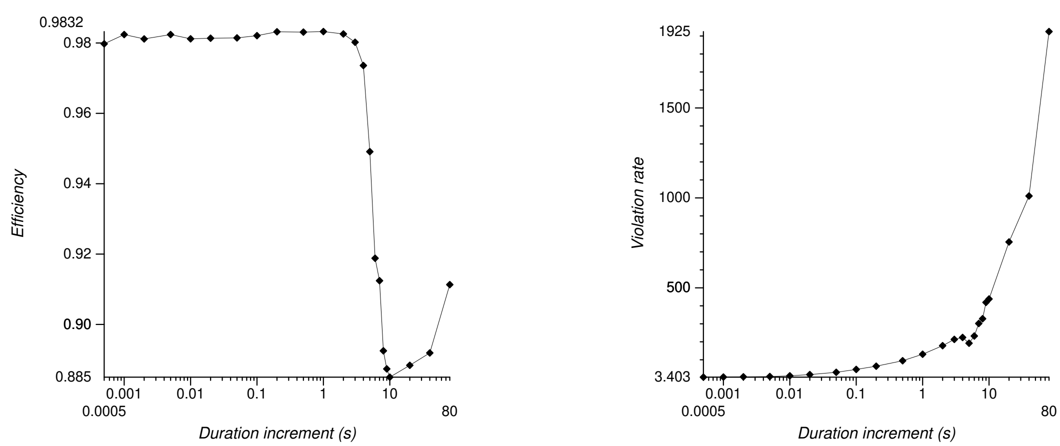

Figure 20: efficiency and violation rate versus BackoffArithGeom duration predictor increment on cell0-disk6.

The violation rate measure showed similar trends. The violation rate remained low for small duration increments, then became much worse at increments larger than 1 or 2 seconds.

There are two behaviors of the adaptive Backoff duration predictors that explain these measures. First, if the increment is too large the predictions oscillate as the predictor increases too much, then backs off to a more reasonable amount—then increases again when the prediction is just shorter than it should be. Second, a large increment leads to long predictions, which in turn cause the prediction to be adjusted less often, leading to poorer-quality predictions.

5.5 Using skeptics

Skeptics filter the stream of idle predictions, hopefully to improve them. To examine whether this was a useful idea or not, we implemented two skeptics that attempted to reduce the violation rate produced by a detector, and obtained the resulting internal measures. While most detectors work to make good predictions in the short term, these skeptics attempted to apply longer-term information.

The skeptics were:

- SkepticTOD (or time-of-day skeptic) only allowed predictions for the least-busy hours of the day. This is an *environmental* skeptic in our taxonomy. It maintained an hour-by-hour moving average of the number of operations requested, and filtered out predictions during the *n*th busiest hours. We experimented with two versions: one that only filtered out the two busiest hours (actually the 90th percentile; labelled TODSensitive below) and one that filtered out the seven busiest hours (70th percentile; TOD below).
- SkepticRate maintained a moving-average estimate of the recent violation rate of the incoming prediction stream, and only passed predictions when the rate estimate was below some threshold. We used a threshold of 0.7 violations/second.

We expected that applying a skeptic would reduce both efficiency and violation rate because the skeptic was filtering out some idle periods. To measure this effect, we repeated the experiment of Section 5.4.2 for tuning predictor parameters, with skeptics inserted into the detector network. We measured the resulting efficiency and violation rate.

The skeptics were occasionally able to improve the violation rate. The TOD skeptics had little effect, as seen in Figure 21, though they never increased the rate. The rate skeptic, on the other hand, improved the violation rate on the cello-disk6 workload significantly for those detectors that showed high rates. For the hplajw-disk0 workload, it sometimes made the violation rate worse, presumably because it filtered out predictions that would have resulted in few violations.

The effects on efficiency were clearer (Figure 22). The TOD skeptics reduced efficiency by 10–20% for all the workloads. The rate skeptic, on the other hand, reduced efficiency by about 70% for hplajw-disk0 and by more than 90% for cello-disk6.

Our conclusion is that skeptics remain an unproven idea. The ones we investigated can help somewhat on a busy system, but on a lightly-loaded system they do not.

6 Conclusions

The high degree of burstiness observed in many real computer systems gives many opportunities for doing useful work at low apparent cost during idle periods. Many people have observed this, and applied this idea to specialized domains.

The contribution of this work is to put the previous approaches into a common framework. This framework should be helpful to those looking to exploit low-utilization periods in computer

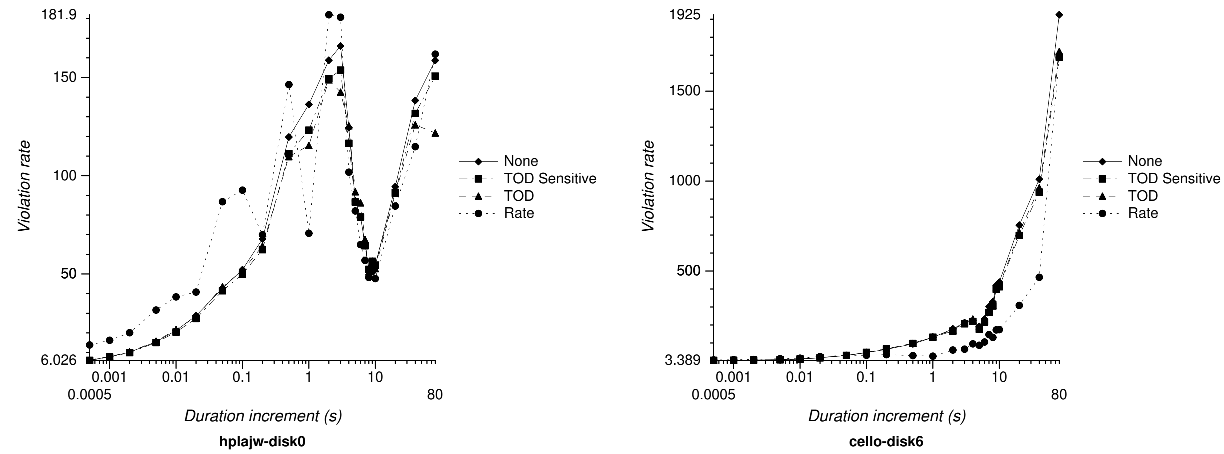

Figure 21: violation rate versus BackoffArithGeom duration predictor increment with skeptics applied.

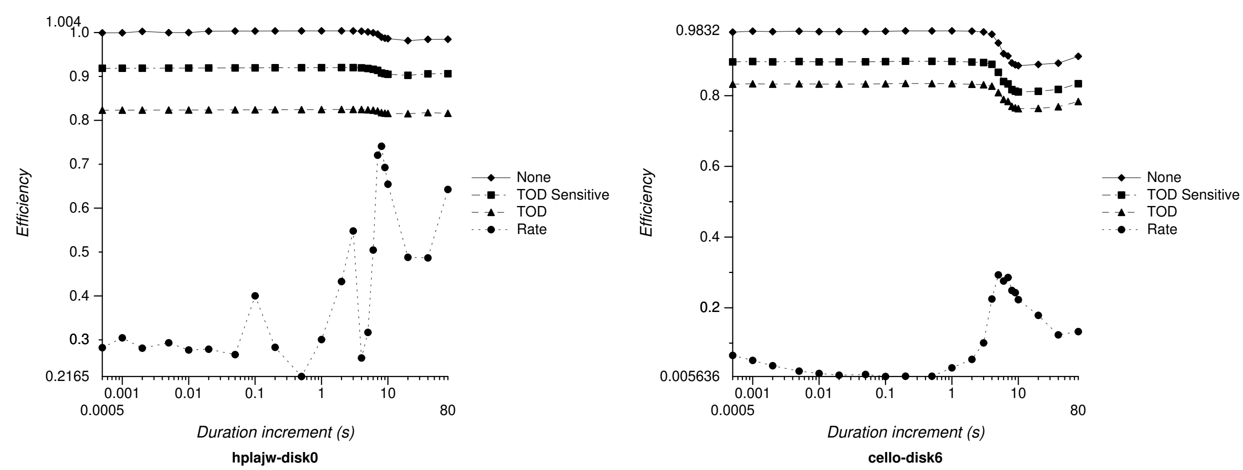

Figure 22: efficiency versus BackoffArithGeom duration predictor increment
with skeptics applied.

systems, regardless of the precise details of the problem, since the framework itself is independent of the particular domain.

Developing the taxonomy was helpful to us in two ways: it improved our understanding of the problem, and it helped us systematize the generation of a large number of potentially interesting detection and prediction algorithms. Without it, we would have had a much harder time exploring the design space.

We had three goals in evaluating the detection mechanisms: to see if they could be applied to realistic tasks and workloads; to find ways to predict which algorithms were appropriate to various tasks by looking for correlations between task performance and task-independent internal measures; and to investigate the sensitivity of some algorithms to their parameters.

We were gratified to learn that simple predictors work remarkably well at our idle tasks. This is good news: it means that these techniques can be applied successfully in the real world with only moderate effort. Rate-based detectors, in particular, work well for idle tasks that can be overlapped with while regular work.

Some detectors and predictors were quite sensitive to their parameters. The quickness with which an EventWindow detector reacted, for example, depended both on the threshold rate and window size.

The self-adapting nature of some of the idleness detectors proved useful, and the adaptive algorithms were often among the best. For some algorithms, the best parameterization values depended on the workload—which could change over time. Rather than have users manually estimate the best value, adaptive mechanisms can automatically adjust to the workload.

The external measures specific to the idle tasks were sometimes related to basic internal measures. For most of the tasks we started with suspicions about how the idle detectors should behave to produce a good result; for example, we were pretty sure that efficiency was important for saving power in the disk spindown task. For the segment cleaning task, on the other hand, while we expected that a low violation rate was important, we did not have intuition about the tradeoff between finding short idle periods often versus long periods more rarely. We discussed the actual relations in Section 5.3.

The implementation of idleness detectors proved straightforward, though we learned a few lessons along the way. We learned that we needed to be sure that adaptive algorithms were able to update their state often to react quickly to changes. We discovered that the notion of continuing a prediction (Section 3.1.3) was useful.

If the foreground work is similar to the storage system traces we used, we can make some general recommendations for picking the best detectors for a task:

- if the idle task completely blocks regular work, or if the cost of interrupting the task is high, then a low violation rate is important. Various kinds of Timer and AdaptTimer detectors work well for this, though the adaptive versions may be more resilient to a wide range of workloads.
- if regular work can continue while the idle task is executing, albeit with some interference, then rate-based detectors, such as the EventWindow and MovingAverage policies, are often a good choice.
- if the granularity of the idle task is large, or the cost of using an inaccurate prediction is high, then prediction duration is important. The adaptive BackoffArithArith and BackoffArithGeom predictors, using a duration increment parameter in the range 0.1 to 1.0 s, often work well.
- the selection of the absolute best tuning parameters isn't usually important, because performance often appears to be stable within a range of parameter values.

6.1 Future directions

We limited our investigation to simple policies, so there are many additions that could be made to this work. Idleness detection is closely related to the general problem of predicting future events, given a series of recent past events. More sophisticated time series analytic approaches could be used for this purpose. There are machine learning techniques that could potentially do even better. We also have wondered about, but did not investigate, the effect of having policies use application-specific feedback, such as how much housekeeping got done during one predicted period.

Our investigation of skeptics was at best cursory, and could easily be extended.

Acknowledgments

George Neville-Neil suggested predictors based on the first derivative of arrival rate, and read several early drafts of this work. Fred Douglis's interest in disk power conservation spurred much of our early thinking. Darrell Long and David Helmbold, at U. C. Santa Cruz, suggested the possibility of using machine learning techniques. The feedback from reviewers and readers of our earlier paper on this subject helped to clarify our thinking and analysis. The other members of the Storage Systems Program at Hewlett-Packard Labs have provided support, encouragement, and comments all during this project.

References

[Baker92b] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12–15 October 1992). Published as *Computer Architecture News*, **20**(special issue):10–22, October 1992.

[Bosch94] Peter Bosch. *A cache odyssey*. M.Sc. thesis, published as Technical Report SPA–94–10. Faculty of Computer Science/SPA, Universiteit Twente, Netherlands, 23 June 1994.

[Box94] George E. P. Box, Gwilym M. Jenkins, and Gregory C. Reinsel. *Time series analysis: forecasting and control*, third edition. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[Bubenik89] Rick Bubenik and Willy Zwaenepoel. Performance of optimistic make. *Proceedings of 1989 ACM SIGMETRICS and Performance '89 International Conference on Measurement and Modeling of Computer Systems* (Berkeley, CA). Published as *Performance Evaluation Review*, **17**(1):39–48, May 1989.

[Cáceres93] Ramón Cáceres, Fred Douglis, Kai Li, and Brian Marsh. *Operating system implications of solid-state mobile computers*. Technical report MITL–TR–56–93. Matsushita Information Technology Laboratory, Princeton, NJ, May 1993.

[Carson92a] Scott Carson and Sanjeev Setia. Optimal write batch size in log-structured file systems. *USENIX Workshop on File Systems* (Ann Arbor, MI), pages 79–91, 21–22 May 1992.

[CesaBianchi94] N. Cesa-Bianchi, Y. Freund, D. P. Helmbold, and M. Warmuth. *On-line prediction and conversion strategies*. Technical report UCSC–CRL–94–28. Computer and Information Sciences Board, University of California at Santa Cruz, August 1994.

[Chen96b] Peter M. Chen. *Optimizing delay in delayed-write file systems*. Technical report CSE–TR–293–96. University Michigan, May 1996.

[Comer91] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP: design, implementation, and internals*, volume II. Prentice-Hall, 1991.

[Cruz92] Rene L. Cruz. Service burstiness and dynamic burstiness measures: a framework. *Journal of High Speed Networks*, **2**:105–27. IOS press, Amsterdam, 1992.

[Douglis95] Fred Douglis, P. Krishnan, and Brian Bershad. Adaptive disk spin-down policies for mobile computers. *Proceedings of Second Usenix Symposium on Mobile and Location-Independent Computing* (Ann Arbor, MI). Usenix Association, 10–11 April 1995.

[Ganger94b] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. *Proceedings of 1st OSDI.* (Monterey, CA), pages 49–60. Usenix Association, 14–17 November 1994.

[Golding95] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Proceedings of Winter USENIX 1995 Technical Conference* (New Orleans, LA), pages 201–12. Usenix Association, Berkeley, CA, 16–20 January 1995.

[Greenawalt94] Paul M. Greenawalt. Modeling power management for hard disks. *2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS '94)* (Durham, NC), pages 62–6. IEEE Computer Society Press, Los Alamitos, CA, 31 January–2 February 1994.

[Gribble96] Steven D. Gribble, Gurmeet Singh Manku, and Eric A. Brewer. Self-similarity in file systems: measurement and applications, 1996. Available at http://www.cs.berkeley.edu/ gribble/cs262/project/project.html.

[Helmbold96] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A dynamic disk spin-down technique for mobile computing. *Proceedings of 2nd Annual International Conference on Mobile Computing and Networking (MOBICOM)* (Rye, NY), 10–12 November 1996.

[HPKittyhawk92] Hewlett-Packard Company, Boise, Idaho. *HP Kittyhawk Personal Storage Module: product brief*, Part number 5091–4760E, 1992.

[Jacobson91] David M. Jacobson and John Wilkes. *Disk scheduling algorithms based on rotational position*. Technical report HPL–CSP–91–7. Hewlett-Packard Laboratories, 24 February 1991, revised 1 March 1991.

[Karn91] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. *ACM Transactions on Computer Systems*, **9**(4):364–73, November 1991.

[Lindsey81] William C. Lindsey and Chak Ming Chie. A survey of digital phase-locked loops. In William C. Lindsey, editor, *Phase Locked Loops*, pages 296–317. Institute of Electrical and Electronics Engineers, April 1981.

[Marsh93] Brian Marsh, Fred Douglis, and P. Krishnan. *Flash memory file caching for mobile computers*. Technical report MITL–TR–59–93. Matsushita Information Technology Laboratory, Princeton, NJ, 18 June 1993.

[Massalin89a] Henry Massalin and Calton Pu. Fine-grain scheduling. *Proceedings of Workshop on Experience in Building Distributed and Multiprocessor Systems* (Ft. Lauderdale, FL), pages 91–104. USENIX Association, October 1989.

[Postel80a] J. Postel. *Transmission Control Protocol*, Technical report RFC–761. USC Information Sciences Institute, January 1980.

[Rodeheffer91] Thomas L. Rodeheffer and Michael D. Schroeder. Automatic reconfiguration in Autonet. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):183–97, 13–16 October 1991.

[Rosenblum92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52, February 1992.

[Ruemmler93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Proceedings of Winter 1993 USENIX* (San Diego, CA), pages 405–20, 25–29 January 1993.

[Ruemmler94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, **27**(3):17–28, March 1994.

[Seltzer90b] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Proceedings of Winter 1990 USENIX Conference* (Washington, D.C.), pages 313–23, 22–26 January 1990.

[Seltzer93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. *Proceedings of Winter 1993 USENIX* (San Diego, CA, 25–29 January 1993), pages 307–26, January 1993.

[Wilkes92b] John Wilkes. *Predictive power conservation*. Technical report HPL–CSP–92–5. Concurrent Systems Project, Hewlett-Packard Laboratories, 14 February 1992.

[Wilkes95] John Wilkes. *The Pantheon storage-system simulator*. Technical Report HPL–SSP–95–14. Storage Systems Program, Hewlett-Packard Laboratories, Palo Alto, CA, 29 December 1995.